INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Fast algorithms for compressed multi-method dispatch tables generation

Éric Amiel, Éric Dujardin, Éric Simon

N° 2977

September 1996

THÈME 3

Rapport de recherche

ISSN 0249-6399

Fast algorithms for compressed multi-method dispatch tables generation

Éric Amiel,[*] Éric Dujardin,[**] Éric Simon[***]

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet RODIN

Rapport de recherche n° 2977 — September 1996 — 63 pages

Abstract: The efficiency of dynamic dispatch is a major impediment to the adoption of multi-methods in object languages. In this paper, we propose a simple multi-method dispatch scheme based on compressed dispatch tables. This scheme is applicable to most existing object languages, and guarantees that dynamic dispatch is performed in constant time, a major requirement for some languages and applications. We provide efficient algorithms to build the compressed dispatch tables, and demonstrate the effectiveness of our scheme by real measurements performed on a large object-oriented application. Finally, we relate our scheme to existing techniques, including a detailed comparison with a recent proposal.

Key-words: multi-methods, late binding, run-time dispatch, dispatch tables, pole types, optimization.

(Résumé : tsvp)

This work was done while the authors were at INRIA.

[*]NatSoft, Air Center, 1214 Genève, Suisse. Email : eamiel@natsoft.ch
[**]Bell Laboratories, Murray Hill, NJ 07974-0636, USA. Email : dujardin@research.bell-labs.com
[***]Email : eric.simon@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Algorithmes efficaces pour la compression des tables d'envoi de multi-méthodes

Résumé : L'efficacité de la sélection dynamique est un obstacle majeur à l'adoption des multi-méthodes dans les langages à objets. Dans ce rapport, nous proposons un système simple de sélection des multi-méthodes, basé sur des tables de sélection comprimées. Ce système peut s'appliquer à la plupart des langages à objets, et garantit que la sélection dynamique s'effectue en temps constant, ce qui est une exigeance majeure pour certains langages et certaines applications. Nous fournissons des algorithmes efficaces de construction des tables d'envoi comprimées, et démontrons l'efficacité de notre système par des mesures réelles effectuées sur une large application à objets. Enfin, nous comparons notre systèmes aux techniques existantes, en le comparant notamment à une proposition récente.

Mots-clé : Multi-méthodes, liaison tardive, sélection dynamique, tables de sélection, types pôles, optimisation.

1 Introduction

In traditional object-oriented systems such as Smalltalk [GR83] and C++ [ES92], functions have a specially designated *target* argument, sometimes called receiver, whose type, determined either at run-time or at compile-time, serves to select the method to execute in response to a function invocation. Multiple dispatching, first introduced in CommonLoops [BKK$^+$86] and CLOS [BDG$^+$88], generalizes this form of dynamic binding by selecting a method depending on the run-time type of a subset of the arguments of a function invocation. Methods in this generalized scheme are called multi-targetted methods, or multi-methods for short. As explained in [CL95], multi-methods bring an increased expressive power over single-targetted methods. For this reason, multi-methods are becoming a key feature of recently developed object-oriented languages as Polyglot [ADL91], Kea [MHH91], Cecil [Cha92], and Dylan [App95]. They also have been integrated as part of the SQL3 standard [Mel96, Mel94] currently under development.

However, several problems still hamper the wide acceptance of multi-methods in object-oriented systems, among which the inefficiency of multiple dispatching caused by the multiple selection criteria of methods. By contrast, there exist algorithmical solutions that offer a fast constant-time dispatching for mono-methods using a two-dimensional table, called dispatch table, that holds the precalculated result of dispatching for all possible function invocations. Optimization techniques have been developed to minimize the size of the dispatch table by eliminating entries corresponding to function invocations for which there is no applicable method [DMSV89, ES92, HC92, DH95]. The reduction factor is experimentally measured to be up to 66 in [DH95]. The dispatch table technique does not naturally scale up for multi-methods because additional target arguments create new dimensions in the table. Since each dimension is indexed by the set of types, which is usually large (e.g., above 100), the size of the table dramatically increases and cannot be handled anymore. Furthermore, since the number of target arguments varies between functions, there must be one dispatch table per function.

The major contribution of this report is to propose a simple, rigorously defined, multi-method dispatch scheme based on compressed dispatch tables that guarantees a dynamic dispatching in constant time. The basic idea of the table's compression scheme is first to eliminate entries corresponding to invocations for which there is no applicable method, and second to group identical $n-1$ dimensional rows in a table of dimension n. Although this scheme can achieve a high compression factor, naïve algorithms to group identical rows in a table of dimension n have a worst-case complexity of $O(n \times |\Theta|^{n+1})$ where $|\Theta|$ represents the total number of types in this system. We show that in a compressed dispatch table, each dimension is indexed by a subset of the types, called the *pole types* of this dimension. We then provide fast algorithms to compute pole types, and hence a compressed table's structure, with a worst-case complexity of $O(|m| + |\Theta| + \mathcal{E})$, where $|m|$ is the number of methods asso ted with a generic function m, and $\mathcal{E}$ is the number of edges in the type graph. Finally, we provide fast algorithms to fill up a dispatch table with a complexity of $O(|\Theta|^2 \times n^2 \times K)$, where K is the maximum number of direct supertypes of all types.

We prove the correctness of our algorithms and demonstrate the effectiveness of our compression scheme using a real object-oriented application with multi-methods. Finally, we provide a detailed comparison of our multi-method dispatch scheme with other existing schemes, including the recent proposal of [CT95], which uses a kind of "dispatch trees".

The report is organized as follows. Section 2 introduces preliminary notions and notations on multi-methods and method dispatch. Section 3 presents the problem and an overview of our solution. Section 4 introduces our dispatch table compression scheme, based on a formal definition of pole types, and proves its correctness. Section 5 presents the pole type computation algorithm, while section 6 presents the table fill-up algorithm. Section 7 describes our implementation and provides experimental results. Section 8 is devoted to a presentation of related work. Finally, we conclude in Section 9.

2 Background

We briefly review some terminology mainly introduced in [ADL91]. We denote *subtyping* by $\preceq$. Given two types T_1 and T_2, if $T_1 \preceq T_2$, we say that T_1 is a subtype of T_2 and T_2 is a supertype of T_1. The set of types is denoted Θ. We assume the existence of a relation *isa* between types, such that the subtyping relation $\preceq$ is the transitive closure of *isa*. If T *isa* T',

T is a *direct subtype* of T', and T' is a *direct supertype* of T. Intuitively, *isa* corresponds to the user's declarations of supertypes.

A generic function is defined by its name and its arity. To each generic function m of arity n corresponds a set of methods $m_k(T_k^1, \ldots, T_k^n) \rightarrow R_k$, where T_k^i is the type of the ith formal argument, and where R_k is the type of the result. We call the list of arguments $(T_k^1, \ldots, T_k^n)$ of method m_k the *signature* of m_k. We also define a relation $\preceq$ on signatures, called *precedence*, such that $(T^1, \ldots, T^n) \preceq (T'^1, \ldots, T'^n)$ iff for all i, $T^i \preceq T'^i$. An invocation of a generic function m is denoted $m(T^1, ..., T^n)$, where $(T^1, \ldots, T^n)$ is the signature of the invocation, and the T^i's represent the types of the expressions passed as arguments.

Object-oriented languages support *late binding* of method code to invocations: the method that gets executed is selected based on the run-time type(s) of the target argument(s). This selection process is called *method dispatch.* In traditional object-oriented systems, generic functions have a single specially designated *target* argument – also called *receiver.* In systems that support multi-methods, a subset of the arguments of a generic function are target arguments. From now on, we shall only consider multi-methods, and without a loss of generality we assume that all arguments are target arguments.

Operationally, method dispatch follows a two-step process: first, based on the types of the target arguments, the set of the applicable methods is found:

Definition 2.1 *A method $m_i(T_i^1, \ldots, T_i^n)$ is applicable to a signature $(T^1, \ldots, T^n)$, iff $(T_i^1, \ldots, T_i^n) \succeq (T^1, \ldots, T^n)$.*

Second, a precedence relationship between applicable methods is used to select what is called the *Most Specific Applicable* (MSA) method. Given a signature $s = (T_1, \ldots, T_n)$ and a generic function invocation $m(s)$, if m_i and m_j are applicable to $m(s)$ and, according to a particular method precedence ordering, m_i is more specific than m_j for s, noted $m_i \preceq_s m_j$, then m_i is a closer match for the invocation. In the rest of this report, we assume that for any function invocation, if there is an applicable method, then there always exists a unique MSA method. We call this the Unique Most Specific Applicable (UMSA) property. The basis of method specificity is a precedence relationship called *argument subtype precedence* in [ADL91]:

Definition 2.2 *Let $m_i(T_i^1, \ldots, T_i^n)$ and $m_j(T_j^1, \ldots, T_j^n)$ be two methods associated with a generic function m of arity n, if $(T_i^1, \ldots, T_i^n) \preceq (T_j^1, \ldots, T_j^n)$, then for any method invocation $m(s)$ such that m_i and m_j are applicable to $m(s)$, $m_i \preceq_s m_j$.*

However, such a relationship does not yield a total order in the presence of multiple inheritance or multi-targetting and thus cannot guarantee the UMSA property. Consequently, additional ordering criteria are needed to obtain a total order.

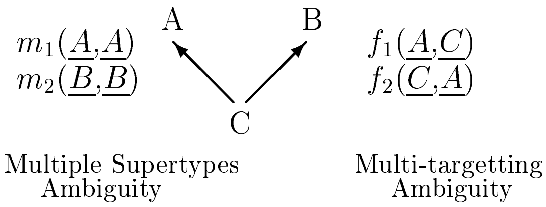


Figure 1: Method Specificity

Example 2.1: Consider the type hierarchy and methods of Figure 1 where the target arguments are underlined. Argument subtype precedence can order neither m_2 w.r.t. m_1 (multiple supertypes ambiguity), nor f_2 w.r.t. f_1 (multi-targetting ambiguity). Thus, argument subtype precedence cannot be used to find the MSA method for invocations $m(C, C)$ and $f(C, C)$. $\square$

Although languages differ in the way they complement argument subtype precedence ordering, our results on method dispatch are applicable to every language that enforces at least argument subtype precedence and *monotonicity*, defined in [DHHM94] as follows.

Definition 2.3 *Let s and s' be two signatures of arity n such that $s \preceq s'$. Let m be a generic function of arity n. If m_k is an applicable method that is not the MSA for $m(s')$ then m_k cannot be the MSA for $m(s)$.*

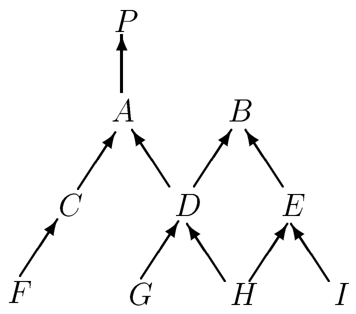
Languages such as Cecil [Cha93] and Dylan [App95] satisfy our required conditions. CLOS [BDG+88] does not enforce monotonicity, as shown in [DHHM92]. [DHHM94] proposes a method precedence algorithm similar to CLOS's that enforces monotonicity.

3 Problem Statement and Overview of the Solution

We address the problem of multi-methods dispatch in languages requiring constant-time method selection. Therefore, our goal is to explore the dispatch table approach both in statically and dynamically typed languages.

3.1 Dispatch Tables for Multi-Methods

As generic functions can have different arities, the single global table organization is not applicable anymore. We need to use a dispatch table for each generic function. For a set of types Θ, the dispatch table of a generic function m of arity n is an n-dimensional array with the types of Θ as indices of each dimension. We denote the dispatch table by $\mathcal{D}_m$. Every entry in $\mathcal{D}_m$ represents a signature in Θ^n. The entry at position $(T_1, \ldots, T_n)$ holds the MSA method for invocation $m(T_1, \ldots, T_n)$ and a null value if there is no MSA. The issue of defining a table entry content (e.g. address, index into a methods array or code) is orthogonal to our purpose (see [Ros88] for a discussion of this issue). In this report, tables contain method's addresses. In our examples, they are replaced by the indices of the methods for clarity and "$-$" denotes the null value.



$m_1(A, B)$
$m_2(A, D)$
$m_3(B, B)$
$m_4(B, C)$
$m_5(B, D)$

Figure 2: Example Schema

Example 3.1: We compute the dispatch table for the type hierarchy and the methods of Figure 2. To this end, we determine the MSA method for every possible invocation using argument subtype precedence and the complementary precedence orders: m_1 is more specific than m_3 and m_2 is more specific than m_5. $\mathcal{D}_m$ is illustrated in Figure 3. □

	P	A	B	C	D	E	F	G	H	I
P	–	–	–	–	–	–	–	–	–	–
A	–	–	1	–	2	1	–	2	2	1
B	–	–	3	4	5	3	4	5	5	3
C	–	–	1	–	2	1	–	2	2	1
D	–	–	1	4	2	1	4	2	2	1
E	–	–	3	4	5	3	4	5	5	3
F	–	–	1	–	2	1	–	2	2	1
G	–	–	1	4	2	1	4	2	2	1
H	–	–	1	4	2	1	4	2	2	1
I	–	–	3	4	5	3	4	5	5	3

Figure 3: Dispatch Table of Generic Function m

Dispatching the methods of a generic function m using a dispatch table is achieved as follows. Assume a unique index is attributed to every type, and every object contains the index of its type as an attribute named *type_index*. Then, in a statically typed language, every invocation $m(o_1, \ldots, o_n)$, where the o_i are objects, is translated into the following pseudo-code:

```
msa = 𝒟_m[o_1.type_index, ..., o_n.type_index]
call msa(o_1, ..., o_n)
```

In a dynamically typed language, an additional test is required to check that the table entry is not empty. In any case, finding the MSA method of an invocation is performed very fast, using a single n-dimensional array access. However, the size of $\mathcal{D}_m$ is $|\Theta|^n$ which is prohibitive ($|\Theta|$ being usually above 100 and the arity of multi-methods between 2 and 4).

3.2 Compressing Dispatch Tables

The core of the problem of multi-method dispatch tables is their size which is in the power of the number of arguments. We propose to reduce the number of entries of each dispatch table and get a compressed table for m that we denote by $\mathcal{D}^c_m$. Assuming a generic function of arity n, our compression scheme is based on two techniques:

- eliminating entries holding only null values.
- grouping entries which have all their values identical.

The following example sketches the idea.

Example 3.2: Consider the table of Figure 3. First, columns P and A, and line P can be eliminated as they only contain null values. This means that invocations where P appears as the first or second argument, or A appears as the second argument are actually type errors. In these cases, examining the type of the argument is sufficient to determine that there is no MSA. Second, lines A, C and F are identical, as are lines B, E and I, and lines D,G and H. The same is true for columns B, E and I, columns C and F, and columns D,G and H. Such rows can be grouped yielding the following table, where indices of the table may be groups of types. Entry at position $(\{T_1, \ldots, T_p\}, \{U_1, \ldots, U_q\})$ holds the MSA method for all invocations $m(T_i, U_j), i \in \{1, \ldots, p\}, j \in \{1, \ldots, q\}$:

	$\{B,E,I\}$	$\{C,F\}$	$\{D,G,H\}$
$\{A,C,F\}$	1	–	2
$\{B,E,I\}$	3	4	5
$\{D,G,H\}$	1	4	2

□

We call the groups of types indexing the dimensions of a compressed table, *index-groups*. In each dimension, index-groups are attributed an index. In our example index-group $\{A,C,F\}$ of the first dimension has index 1. The index of a type in a dimension of a compressed table is the index of its group.

3.3 Dispatch Using Compressed Tables

Once the table is compressed, dispatch must be performed differently. Indeed, the index of a type is not anymore the same in every dimension of a dispatch table and in every dispatch table. Thus, the unique index of each type cannot be directly used to access entries in compressed dispatch tables. In the compressed table of Example 3.2, the index of type A

is 1 (index of its group) when A appears as the first argument, while A does not appear in the second dimension.

We map, for every generic function m and every argument position i, a type to its index in the ith dimension of the compressed dispatch table of m. This can be achieved using n *argument-arrays*, where n is the arity of m. Each argument-array is a one-dimensional array indexed by the types. The ith argument-array of m holds the positions of every type in the ith dimension of the compressed dispatch table of m, i.e., when the type appears as ith argument. A "0" indicates that this type cannot appear as the ith argument.

Example 3.3: Here are the two argument-arrays m_arg_1 and m_arg_2 of m for the compressed dispatch table of Example 3.2:

	P	A	B	C	D	E	F	G	H	I
m_arg_1	0	1	2	1	3	2	1	3	3	2
m_arg_2	0	0	1	2	3	1	2	3	3	1

□

In a statically typed language, a multi-method invocation $m(o_1, \ldots, o_n)$ is translated into the following code:

msa = $\mathcal{D}^c_m[m\_arg_1[o_1.type\_index], \ldots, m\_arg_n[o_n.type\_index]]$
call msa($o_1, \ldots, o_n$)

In a dynamically typed language, run-time type checking is needed. It involves verifying that neither the $m\_arg_i[o_i.index]$ nor the entry of $\mathcal{D}^c_m$ are null before calling the MSA method.

3.4 Argument-Array Coloring

Coloring has been proposed in [DMSV89] to compress dispatch tables of mono-methods. This technique is applicable to argument-arrays for grouping into one array several arrays which hold non null cells at different positions.

We shall call *selector*, noted (m, i), an argument position i in a generic function m. A selector is associated with a set of types, which are the types allowed at run-time for this method and this argument position. Two selectors are said to *conflict* if their sets of types intersect. When two or more selectors do not conflict with each other, their colors can be the same, i.e. their argument-arrays can be grouped, as shown in Figure 4. These arrays do not necessarily belong to the same generic function.

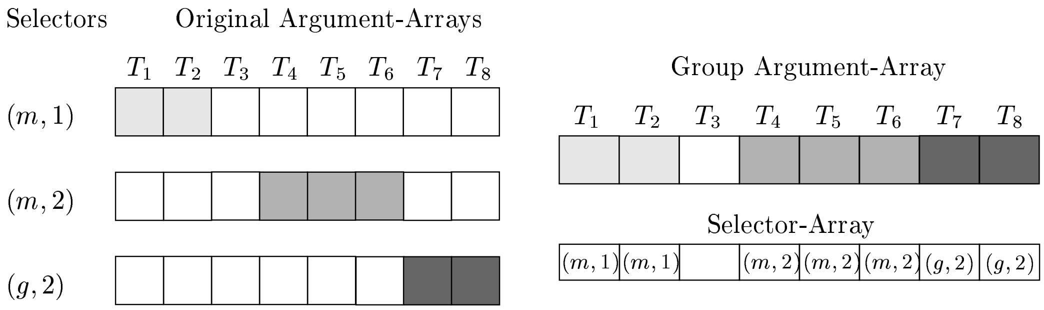


Figure 4: Grouping of Argument-Arrays

When argument-arrays are grouped, the information represented by a null value in the original argument-arrays is lost in the group argument-array. This information is used by run-time type checking, in dynamically typed languages like Smalltalk or Self, or in [ABDS96]. If a type T is not allowed for a selector (m, i), type checking makes it necessary to mark that the entry of T in the group argument-array is related to another selector. For this, we associate with each group argument-array a selector array. For each type, this array holds the selector with which it is associated in the group argument-array. At run-time, the type of the object that appears as i-th argument in an invocation of m, yields both an entry number (coming from the group argument-array) and a selector (coming from the selector array). Type checking consists in comparing the selector to (m, i), and raises an error if they differ. If not, the entry number is used against the dispatch table to find the MSA method.

3.5 Naïve Approach to Compress a Dispatch Table

The naïve approach starts from an uncompressed dispatch table and then compresses it. Building the uncompressed dispatch table simply involves computing the MSA method for every possible invocation signature. With a generic function of arity n, compressing the table comes down to first scan all entries in all dimensions to detect the empty entries and eliminate them, and then compare the remaining entries, to group the ones that are identical.

However, this process is unrealistic, both for memory space and processing time considerations:

1. MSA computation time: assuming an application with 100 types, and the use of a library like the programming environment of Smalltalk which roughly counts 800 more types, there are 810,000 possible invocation signatures for 2-targetted generic functions, and 729,000,000 for 3-targetted generic functions, for which the MSA method has to be computed.

2. Uncompressed table size: with $|\Theta| = 900$, the uncompressed table of a 3-targetted method takes 695 MB, and the one of a 4-targetted method takes 610 GB.

3. Empty entries elimination: searching the empty entries needs at worst to scan n times the entire table, once for each dimension. Moreover, eliminating the empty entries requires to move a lot of data in memory.

4. Identical entries grouping: grouping the entries needs at worst, for each dimension and each entry, to compare it to $|\Theta| - 1$ other entries. Each of these entries is associated with $|\Theta|^{n-1}$ values. Hence grouping takes at worst $n \times |\Theta| \times (|\Theta| - 1) \times |\Theta|^{n-1} = n \times (|\Theta|^{n+1} - |\Theta|^n)$ equality comparisons. For $n = 3$, $|\Theta| = 900$, assuming each comparison is done in one cycle by a 100MHz processor, this nearly takes six hours. Grouping also involves a lot of memory moves.

In the following, we present an efficient algorithm to directly build compressed dispatch tables. For this, we first analyse the set of formal argument types in each dimension, to determine the empty entries and the index groups. Then, we build the compressed dispatch table by computing the MSA method only once for each index-group signature. In this way, both processing time and memory space is spared, and obtaining the compressed dispatch table becomes realistic.

4 Compression Approach

The central issue in our approach to dispatch table compression is to determine which entries can be eliminated and which ones can be grouped without having to scan the whole table. Let us first formally define the condition under which an entry can be eliminated and two entries can be grouped.

Consider the ith argument position of a generic function m of arity n.

Elimination Condition. The entry for type T in the ith dimension of the dispatch table of m can be eliminated iff:

$$\forall(T_1,\ldots,T_{n-1}) \in \Theta^{n-1}, \text{ there is no MSA for } m(T_1,\ldots,T_{i-1},T,T_i,\ldots,T_{n-1}). \quad (1)$$

Grouping Condition. The two entries for types T and T' in the ith dimension of the dispatch table of m can be grouped iff:

$$\forall(T_1,\ldots,T_{n-1}) \in \Theta^{n-1},$$
$$MSA(m(T_1,\ldots,T_{i-1},T,T_{i+1},\ldots,T_n)) = MSA(m(T_1,\ldots,T_{i-1},T',T_{i+1},\ldots,T_n)) \quad (2)$$

The first condition says that T is not allowed as the type of the ith argument of an invocation of m. The second condition says that T can replace T' as the ith-argument of any invocation without any change of the MSA and vice-versa. If this condition holds, then T and T' belong to the same index-group in the ith-dimension and thus have the same index in the ith argument-array of m.

Coming back to Example 3.2, it is useful to observe that in every index-group there is a type supertype of all the other types of the group. For instance, in the groups of the first dimension, A, B and D are respectively the greatest types for groups $\{A,C,F\}$, $\{B,E,I\}$ and $\{D,G,H\}$. We call these types *i-poles*, where i is the argument position. Thus, A, B, and D are 1-poles and B, C and D are 2-poles. We shall show that there always exists a unique i-pole for every group in the i-th dimension. In fact, we show how to determine for every dimension the i-poles and their associated index-groups, called *influence*. Then, we establish that:

- The types that do not belong to an influence can be eliminated from the ith dimension.

- The types belonging to the same influence can be grouped.

4.1 Poles and Influences

We first introduce some auxiliary definitions and then formally define the notions of i-pole and influence.

Definition 4.1 *T_1 and T_2 are incomparable, denoted by $T_1 \prec\succ T_2$, iff:*

$$T_1 \not\preceq T_2 \text{ and } T_1 \not\succeq T_2.$$

Definition 4.2 *The cover of a type T, noted $cover(T)$, is the set of subtypes of T:*

$$cover(T) = \{T' \mid T' \preceq T\}$$

The cover of a set of types $\{T_1, \ldots, T_n\}$, noted $cover(\{T_1, \ldots, T_n\})$, is the union of the covers of each type in the set:

$$cover(\{T_1, \ldots, T_n\}) = \bigcup_{i=1}^{n} cover(T_i)$$

Definition 4.3 *The ith static arguments of a generic function m, noted $Static_m^i$, are the types of the ith formal arguments of the methods of m:*

$$Static_m^i = \{T \mid \exists m_k \text{ with } T_k^i = T\}$$

Definition 4.4 *The ith dynamic arguments of a generic function m, noted $Dynamic_m^i$, are the cover of the ith static arguments of m:*

$$Dynamic_m^i = cover(Static_m^i)$$

They represent the types that can appear at the ith position in invocations of m at run-time. $Dynamic_m$ is the cross product of the ith dynamic arguments sets:

$$Dynamic_m = \prod_{i=1}^{n} Dynamic_m^i$$

Fact 4.1 *Every signature for which there is an MSA method belongs to $Dynamic_m$.*

Definition 4.5 *A type $T \in \Theta$ is an i-pole of a generic function m of arity n, $1 \leq i \leq n$, iff:*

$$T \in Static_m^i \tag{3}$$

$$\text{or} \quad |min_{\preceq} \ \{T' \in \Theta \mid T' \text{ is an } i\text{-pole of } m \text{ and } T' \succ T\}| > 1 \tag{4}$$

The set of i-poles of m is denoted $Pole_m^i$. The poles that are included in $Static_m^i$ are called *primary poles*, and the others are called *secondary poles*.

Definition 4.6 *Let m be a generic function of arity n, $T \in \Theta$, and $1 \leq i \leq n$, the set of closest poles of T is:*

$$closest\text{-}poles_m^i(T) = min_{\preceq}\{T' \in Pole_m^i \mid T \prec T'\}$$

Using this notation, condition (4) can also be expressed as:

$$|closest\text{-}poles_m^i(T)| > 1 \tag{5}$$

Lemma 4.1 *Given a generic function m of arity n, for each i, $1 \leq i \leq n$ and each $T \in \Theta$, we have:*

$$closest\text{-}poles_m^i(T) \neq \emptyset \Rightarrow T \in Dynamic_m^i$$

Proof: If $closest\text{-}poles_m^i(T) \neq \emptyset$ then there exists some $T' \in Pole_m^i$ such that $T \preceq T'$. To establish the lemma, we show that there exists $T'' \in Static_m^i$ such that $T \preceq T' \preceq T''$, i.e., $T \in cover(T'')$. We proceed by contradiction. Suppose

$$\forall T' \in Pole_m^i(T) \text{ s.t. } T \preceq T', \ \nexists T'' \in Static_m^i \text{ s.t. } T' \preceq T''. \tag{6}$$

In particular $T' \notin Static_m^i$. Since T' is an i-pole, by condition (4) $\exists T_0 \in Pole_m^i$ such that $T' \preceq T_0$. Hence $T \preceq T_0$, and by (6), $T_0 \notin Static_m^i$. Iterating the same reasoning, we can prove the existence of an infinite sequence of distinct poles $T_0, T_1, \ldots$, which contradicts the fact that Θ is finite. $\square$

Proposition 4.1 *Given a generic function m of arity n, and $i \in \{1, \ldots, n\}$, we have:*

$$(\forall T' \in Static_m^i,\ \forall T \in \Theta \text{ s.t. } T \prec T',\ T \text{ has exactly one supertype}) \Rightarrow (Pole_m^i = Static_m^i).$$

Proof: We have to show that $Pole_m^i \subset Static_m^i$, i.e. $\forall T \in \Theta$, T is not a secondary pole. Let $T \in \Theta$. If $closest\text{-}poles_m^i(T) = \emptyset$ then $\{T' \in Static_m^i \mid T \prec T'\} = \emptyset$ and the proposition trivially holds. If $closest\text{-}poles_m^i(T) \neq \emptyset$ then by Lemma 4.1, $T \in Dynamic_m^i$. Thus, by definition, there exists $T' \in Static_m^i$ such that $T \in cover(T')$. We show that $|closest\text{-}poles_m^i(T)| = 1$, which by condition (5) means that T is not a secondary pole. We proceed by contradiction.

Let T_a and T_b be two distinct poles of $closest\text{-}poles_m^i(T)$. Let $T_1, \ldots, T_k$ such that $T\ isa\ T_1 \ldots\ isa\ T_k\ isa\ T'$. By left-hand side of the proposition, each type $T, T_1, \ldots, T_k$ has only one direct supertype. Since $\{T, T_1, \ldots, T_k, T'\}$ is totally ordered with respect to $\preceq$ and T_a and T_b are incomparable by definition, this set does not include both T_a and T_b. Suppose that T_b is not in this set. By left-hand side of the proposition, the path from T to T_b in the type graph necessarily goes through T', hence $T' \preceq T_b$. As $T_b \notin \{T, T_1, \ldots, T_k, T'\}$, $T' \prec T^b$.

Since $T' \in Static_m^i$, T' is an i-pole, and by Definition 4.6, $T_b \preceq T'$, a final contradiction. □

Corollary 4.1 *In systems supporting or actually using only single inheritance, for each generic function m, and each i in $\{1, \ldots, n\}$, $Pole_m^i = Static_m^i$.*

Example 4.1: We determine the 1- and 2-poles in the example of Figure 2. We have $Static_m^1 = \{A, B\}$, $Static_m^2 = \{B, C, D\}$, $Dynamic_m^1 = \{A, C, D, F, G, H\}$, and $Dynamic_m^2 = \{B, C, D, E, F, G, H, I\}$. As $Static_m^1 = \{A, B\}$, A and B are 1-poles. Moreover, they are the closest poles of D, so D is also a 1-pole. So $Pole_m^1 = \{A, B, D\}$. As $Static_m^2 = \{B, C, D\}$, B, C and D are 2-poles. They do not satisfy condition (3) w.r.t. any type, so $Pole_m^2 = \{B, C, D\}$. The 1-poles are underlined on Figure 5. □

Definition 4.7 *With every i-pole T of a generic function m is associated the set of subtypes of T, noted $Influence_m^i(T)$, defined as:*

$$Influence_m^i(T) = \{T' \preceq T \mid \forall T'' \in Pole_m^i, T' \not\preceq T'' \text{ or } T \preceq T''\}.$$

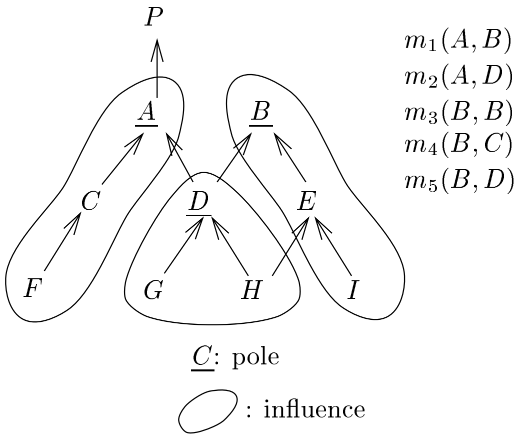


Figure 5: Example Schema with Poles and Influences

Example 4.2: We determine $Influence^i_m$ for the 1-poles of Example 4.1. The influence of A contains A, C and F as A is their only 1-pole supertype. For the same reason, the influence of B contains B, E and I. D does not belong to the influence of either A or B as D is a 1-pole, and it is supertype of itself and a subtype of A and B. The influence of D contains D, G and H as D is the single closest 1-pole. The influences are surrounded by a blob on Figure 5. □

Proposition 4.2 *Given a generic function m of arity n, and $i \in \{1, \ldots, n\}$, Let $Pole^i_m = \{T_1, \ldots, T_l\}$, then:*

1. *$\{Influence^i_m(T_1), \ldots, Influence^i_m(T_l)\}$ is a partition of $Dynamic^i_m$, and*
2. *$\forall T \in Dynamic^i_m$, $(T \in Influence^i_m(T_k) \Leftrightarrow min_{\preceq}\{T' \in Pole^i_m \mid T \preceq T'\} = \{T_k\})$*

Proof:

1a. We first show that influences are pairwise disjoint. We proceed by contradiction. Suppose that T_1 and T_2 are in $Pole^i_m$, and $T \in Influence^i_m(T_1) \cap Influence^i_m(T_2)$. Hence, $T \preceq T_1$ and $T \preceq T_2$. By definition of $Influence^i_m(T_1)$ and $T \preceq T_2$, we infer that $T_1 \preceq T_2$, and symmetrically that $T_2 \preceq T_1$. Therefore $T_1 = T_2$, a contradiction.

1b. We now show that: $\bigcup_{k \preceq l} Influence^i_m(T_k) = Dynamic^i_m$.

$\subseteq$: If $T \in Incluence^i_m(T_k)$ then $closest\text{-}poles^i_m(T) = \{T_k\}$, and by Lemma 4.1, $T \in Dynamic^i_m$.

$\supseteq$: Let $T \in Dynamic^i_m$. If T is a pole then $T \in Influence^i_m(T)$. Otherwise, $|closest\text{-}poles^i_m(T)| \leq 1$, by condition (4). Since $T \in Dynamic^i_m$, there exists an i-pole T' such that $T \in cover(T')$. Thus, $|closest\text{-}poles^i_m(T)| > 0$. Then, there exists T' such that $\{T'\} = closest\text{-}poles^i_m(T)$, and by definition 4.7, $T \in Influence^i_m(T')$.

2. $\Leftarrow$: This is shown by what precedes.

$\Rightarrow$: If $T \in Influence^i_m(T_k)$ then let $\{T'_k\} = min_{\preceq}\{T_p \in Pole^i_m \mid T \prec T_p\}$. By assertion (1), $T_k = T'_k$.

□

As a consequence, each type of $Dynamic^i_m$ is in the influence of one and only one pole of $Pole^i_m$. We use this to define $pole^i_m$.

Definition 4.8 *For each type T in Θ, we define $pole^i_m$:*

- $\forall T \in Dynamic^i_m,\ pole^i_m(T) = T' \iff T \in Influence^i_m(T')$
- $\forall T \in \Theta - Dynamic^i_m,\ pole^i_m(T) = 0$

From this definition, and Proposition 4.2, we have:

Corollary 4.2 *Given a generic function m of arity n, $i \in \{1, \ldots, n\}$, for each type $T \in Dynamic^i_m$,*

$$min_{\preceq}\{T' \in Pole^i_m \mid T \preceq T'\} = \{pole^i_m(T)\}$$

4.2 Main Results

The following two theorems respectively establish that in the ith dimension, the entries for types which do not belong to any influence can be eliminated and the entries for types belonging to the same influence can be grouped.

Theorem 1 *For every generic function m of arity n, and $i \in \{1, \ldots, n\}$, the entry for type T can be eliminated from the dimension i of the dispatch table of m if T does not belong to the influence of any i-pole.*

Proof: By condition (1), we have to prove that if T does not belong to the influence of any i-pole, then it verifies:

$$\forall (T_1, \ldots, T_{n-1}) \in \Theta^{n-1}, MSA(m(T_1, \ldots, T_{i-1}, T, T_{i+1}, \ldots, T_n)) \text{ does not exist.}$$

By assertion 1 of Proposition 4.2, $T \notin Dynamic^i_m$. Thus, there is no $T' \in Static^i_m$ such that $T \preceq T'$. Hence, by definition 4.3, there is no method $m_k(T^1_k, \ldots, T^n_k)$ such that $T \preceq T^i_k$. By definition 2.2, $\forall (T^1, \ldots, T^{n-1}) \in \Theta^{n-1}$, there is no method applicable to $m(T^1, \ldots, T^{i-1}, T, T^i, \ldots, T^{n-1})$. $\Box$

Lemma 4.2 *Given a generic function m of arity n, for all $(T_1, \ldots, T_n)$ in $Dynamic_m$,*

$$MSA(m(T_1, \ldots, T_n))) = MSA(m(pole^1_m(T_1), \ldots, pole^n_m(T_n)))$$

Proof:

- If there is no method applicable to $(T_1, \ldots, T_n)$, as $(pole^1_m(T_1), \ldots, pole^n_m(T_n)) \succeq (T_1, \ldots, T_n)$, from Definition 2.1, no method is applicable to $(pole^1_m(T_1), \ldots, pole^n_m(T_n))$ either. Hence both MSA are null.

- If there is a method applicable to $(pole^1_m(T_1), \ldots, pole^n_m(T_n))$, let $m_k(T^1_k, \ldots, T^n_k) = MSA(m(T_1, \ldots, T_n))$. We first show that m_k is applicable to $(pole^1_m(T_1), \ldots, pole^n_m(T_n))$. Since m_k is applicable to $(T_1, \ldots, T_n)$, from Definition 2.1, for all i, $1 \leq i \leq n$, $T_i \preceq T^i_k$. By condition (3), T^i_k is an i-pole. By assertion 1 of proposition 4.2, for each i, $1 \leq i \leq n$, there exists T'_i such that $T_i \in Influence^i_m(T'_i)$. By definition 4.7, and the facts that $T^i_k \in Pole^i_m$ and $T_i \preceq T^i_k$, $T'_i \preceq T^i_k$.

  We now proceed by contradiction. Suppose that m_k is different from $m'_k = MSA(m(T'_1, \ldots, T'_n))$. Since m_k is not the MSA method for $m(T'_1, \ldots, T'_n)$ and $(T_1, \ldots, T_n) \preceq (T'_1, \ldots, T'_n)$, by Definition 2.3, m_k cannot be the MSA for $m(T_1, \ldots, T_n)$, a contradiction.

□

Theorem 2 *Let m be a generic function of arity n, for each i, $1 \leq i \leq n$, and for each type $T \in Dynamic^i_m$, the entries for type T and $pole^i_m(T)$ can be grouped in the dispatch table.*

Proof: By Condition (2), we have to prove that for each i, $1 \leq i \leq n$, $\forall T \in Dynamic^i_m$, $\forall (T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_n) \in \Theta^{n-1}$,

$$MSA(m(T_1, \ldots, T_{i-1}, T, T_{i+1}, \ldots, T_n)) = MSA(m(T_1, \ldots, pole^i_m(T), \ldots, T_n)) \qquad (7)$$

For each $j \neq i$, if $T_j \notin Dynamic^j_m$ then both MSA methods in (7) do not exist. If $T_j \in Dynamic^j_m$, for each j, $1 \leq j \leq n$, then by Lemma 4.2,

$$MSA(m(T_1, \ldots, T_n)) = MSA(m(pole^1_m(T_1), \ldots, pole^i_m(T), \ldots, pole^n_m(T_n)))$$

and

$$\begin{aligned} & MSA(m(T_1, \ldots, pole^i_m(T), \ldots, T_n)) \\ = \quad & MSA(m(pole^1_m(T_1), \ldots, pole^i_m(pole^i_m(T)), \ldots, pole^n_m(T_n))) \end{aligned}$$

For each i, $1 \leq i \leq n$, for each $T \in Pole^i_m$, by Definition 4.7, $T \in Influence^i_m(T)$. Hence, $pole^i_m(pole^i_m(T)) = pole^i_m(T)$, which proves (7). □

4.3 General Algorithm

From the above definitions and theorems, the general algorithm to compute the compressed dispatch table of a generic function m is the following:

1. Compute the i-poles and their influence, for each selector (m, i).
2. Attribute an index to each i-pole.
3. Compute the selector conflicts, define the color of each selector, and give an identifier to each selector of the same color.
4. For every type T and each argument position i, store the index of its i-pole in the entry of T in the group argument array associated with selector (m, i), and store the selector number at the same position in the corresponding selector array.

5. For every signature $(T_1, \ldots, T_n) \in \prod_{i=1}^{n} Pole_m^i$, determine $MSA(m(T_1, \ldots, T_n))$ and store the result (address or index of the method) in the corresponding entry of the table, i.e., in $\mathcal{D}_m^c[m\_arg_1[T_1.type\_index], \ldots, m\_arg_n[T_n.type\_index]]$.

5 Computing Poles

Computing the i-poles and influences of a generic function m amounts to successively compute the pole of each type in Θ, i.e., $pole_m^i$. This is needed because T is a pole iff $pole_m^i(T) = T$, and if T' is a pole then $Influence(T') = \{T \in \Theta \mid pole_m^i(T) = T'\}$. Moreover, the values of $pole_m^i$ are also needed to build the argument-arrays, whereas the values of $Influence$ are not needed in the general algorithm given above.

We show that the computation of $pole_m^i$ can be done in a single pass over the set of types using $closest\text{-}poles_m^i$. We then present two techniques to optimize the computation of $closest\text{-}poles_m^i$, first by minimizing the set in which they need to be searched, then by using bit vectors to efficiently represent the subtyping relationship.

5.1 Single-Pass Traversal of the Type Graph

We consider a generic function m of arity n, and an argument position i. We show that $pole_m^i(T)$ only depends on $closest\text{-}poles_m^i(T)$.

Theorem 3 *Given a generic function m of arity n, for each i, $1 \leq i \leq n$, for each T in Θ,*

1. $(pole_m^i(T) = 0) \Leftrightarrow (T \notin Static_m^i$ *and* $closest\text{-}poles_m^i(T) = \emptyset)$.
2. $(pole_m^i(T) = T',\ T' \neq T) \Leftrightarrow (T \notin Static_m^i$ *and* $closest\text{-}poles_m^i(T) = \{T'\})$
3. $(pole_m^i(T) = T) \Leftrightarrow (T \in Static_m^i$ *or* $|closest\text{-}poles_m^i(T)| > 1)$

Proof: We first show that

$$T = pole_m^i(T) \Leftrightarrow T \in Pole_m^i \tag{8}$$

If $T = pole^i_m(T)$, from Definition 4.8, $T \in Influence^i_m(T)$, and by Definition 4.7, $T \in Pole^i_m$. Conversely if $T \in Pole^i_m$, as $T \in \{T' \in Pole^i_m \mid T \preceq T'\}$, by Corollary 4.2 $T = pole^i_m(T)$.

From (3), $T \in Static^i_m \Rightarrow T \in Pole^i_m$, hence by (8), $pole^i_m(T) = T$. Thus, we have:

$$T \in Static^i_m \Rightarrow pole^i_m(T) = T \tag{9}$$

We finally have the following fact:

$$T \notin Pole^i_m \Rightarrow \{T' \in Pole^i_m \mid T \prec T'\} = \{T' \in Pole^i_m \mid T \preceq T'\} \tag{10}$$

1. $\Rightarrow$: As $pole^i_m(T) \neq T$, from (9) $T \notin Static^i_m$. Suppose that $closest\text{-}poles^i_m(T) \neq \emptyset$, then by Lemma 4.1, $T \in Dynamic^i_m$ and by Definition 4.8, $pole^i_m(T) \neq 0$.

   $\Leftarrow$: From Definition 4.5, $Static^i_m \subseteq Pole^i_m$, hence $closest\text{-}poles^i_m(T) = \emptyset$ implies $\{T' \in Static^i_m \mid T \prec T'\} = \emptyset$. As $T \notin Static^i_m$, from Definition 4.4, $T \notin Dynamic^i_m$. Using Definition 4.8, this implies $pole^i_m(T) = 0$.

2. $\Rightarrow$: As $pole^i_m(T) \neq T$, from (9), $T \notin Static^i_m$. From Corollary 4.2 we have $\{T'\} = min_{\preceq}\{T'' \in Pole^i_m \mid T \preceq T''\}$. From (8), $T \notin Pole^i_m$, hence (10) applies and from the definition of $closest\text{-}poles^i_m$, $\{T'\} = closest\text{-}poles^i_m(T)$.

   $\Leftarrow$: From Lemma 4.1, $T \in Dynamic^i_m$, hence from Corollary 4.2, $\{pole^i_m(T)\} = min_{\preceq}\{T' \in Pole^i_m \mid T \preceq T'\}$. From (3) and (5), $T \notin Pole^i_m$. Hence (10) applies and $\{pole^i_m(T)\} = min_{\preceq}\{T' \in Pole^i_m \mid T \prec T'\}$. Then by definition of T' and of $closest\text{-}poles^i_m$, $pole^i_m(T) = T'$.

3. $\Rightarrow$: From (8) $T \in Pole^i_m$, hence by definition 4.5 and (5), $T \in Static^i_m$ or $|closest\text{-}poles^i_m(T)| > 1$.

   $\Leftarrow$: If $T \in Static^i_m$, from (9), $pole^i_m(T) = T$. If $|closest\text{-}poles^i_m(T)| > 1$, from (5), $T \in Pole^i_m$ and from (8), $pole^i_m(T) = T$.

□

```
input      : an ordered set of types Θ≤, a generic function m, an argument position i
output     : a set of poles Poles
side-effect: updated pole attribute of the types

Poles ←∅
for T in (Θ, ≤) do
   if T ∈ Static^i_m then                  // case 3a of Theorem 3
      T.pole ← T                           // T is a primary pole
      insert T into Poles
   else
      if closest-poles^i_m(T) = ∅ then     // case 1 of Theorem 3
         T.pole ← 0
      else
         if closest-poles^i_m(T) = {T_p} then   // case 2 of Theorem 3
            T.pole ← T_p
         else                              // case 3b of Theorem 3
            T.pole ← T                     // T is a secondary pole
            insert T into Poles
```

Figure 6: Pole Computation Algorithm

We define now the linear order $\leq$ over Θ, which is used by our algorithm to compute the poles. As $\preceq$ is a partial order over Θ, it is possible to find a total order α over Θ that is an extension of $\preceq$, i.e.

$$(\forall T, T' \in \Theta,\ (T\ \alpha\ T' \text{ or } T'\ \alpha\ T)) \text{ and } (\forall T, T' \in \Theta,\ (T \preceq T' \Rightarrow T\ \alpha\ T'))$$

The topological sort of [Knu73] is a classical algorithm to obtain a linear extension from a partial order. We define $\leq$ to be the opposite of a linear extension of $\preceq$, hence the highest types according to $\preceq$ are the first according to $\leq$:

$$(\forall T, T' \in \Theta,\ (T \leq T' \text{ or } T' \leq T)) \text{ and } (\forall T, T' \in \Theta,\ (T \preceq T' \Rightarrow T' \leq T))$$

We note $(\Theta, \leq)$ the set of all types totally ordered by $\leq$. Our algorithm starts from the highest types and traverses the type graph downward to the most specific types. Thus, each type is treated before its subtypes and after its supertypes. Since the closest poles of T only involve poles that are supertypes of T, and all the supertypes of T have already been treated before T, all poles supertypes of T are known when T is treated. Thus, the body of the algorithm merely implements the case analysis given in Theorem 3.

In the algorithm of Figure 6, we use a record-like representation of types, with an attribute *pole*. A variable *Poles* records the set of poles.

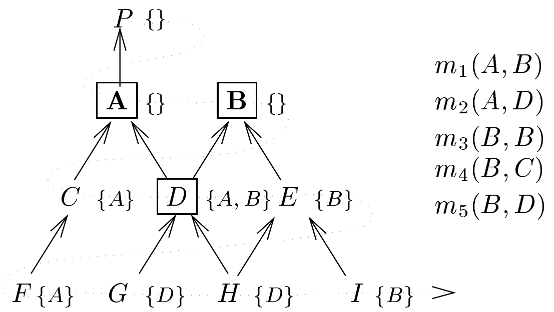


Figure 7: Pole Computation Example

Example 5.1: Figure 7 illustrates the computation of the 1-poles of method m. Poles appear in boxes, and primary poles are written in bold. The dashed arrow shows the order in which types are considered. The closest poles for each type are given between curly brackets. The closest poles of D are A and B, thus D is a pole. The poles supertypes of H are D and B, and as D is a subtype of A, the single closest pole of H is D. Hence H is not a pole and its pole is D. $\square$

5.2 Computing the Closest Poles

The computation of the closest poles can be optimized by reducing the number of poles that need to be compared at each step. The following theorem reduces this number of comparisons to the number of direct supertypes of the current type.

Theorem 4 *For all T in Θ, we have:*

$$closest\text{-}poles^i_m(T) = min_{\preceq}\{pole^i_m(T') \mid T \ isa \ T'\} \tag{11}$$

Proof: Let $T_1, \ldots, T_N$ be the totally ordered list of types in $(\Theta, \leq)$. The proof is by induction on k, $1 \leq k \leq N$.

Basis of the induction: T_1 has no supertype, thus $\{pole^i_m(T') \mid T_1 \ isa \ T'\} = \emptyset$, and since $closest\text{-}poles^i_m(T_1) = \emptyset$, (11) holds.

Claim: Let $(S, \preceq)$ be a partially ordered set, if A and B are two subsets of S,

$$((A \subset B) \text{ and } (min_{\preceq}B \subset A)) \;\Rightarrow (min_{\preceq}A = min_{\preceq}B). \tag{12}$$

Proof: First, let A and B be two subsets of S,

$$A \subset B \Rightarrow \forall x \in min_{\preceq}B,\; \nexists y \in min_{\preceq}A \text{ s.t. } y \prec x. \tag{13}$$

To see that, suppose that $x \in min_{\preceq}B$ and $y \in min_{\preceq}A$. As $A \subset B$, $y \in B$. If $y \prec x$ then $x \notin min_{\preceq}B$, a contradiction.

Suppose that $A \subset B$, and $min_{\preceq}B \subset A$.

$\subseteq$: Let $x \in min_{\preceq}A$. As $A \subset B$, $x \in B$. Since $min_{\preceq}B \subset A$, by (12), $\nexists y \in min_{\preceq}(min_{\preceq}B)$, $y \prec x$, i.e., $\nexists y \in min_{\preceq}B,\; y \prec x$. Hence, $x \in min_{\preceq}B$.

$\supseteq$: Let $x \in min_{\preceq}B$. As $min_{\preceq}B \subset A$, $x \in A$. Since $A \subset B$, by (12), $\nexists y \in min_{\preceq}A,\; y \prec x$. Hence, $x \in min_{\preceq}A$.

□

We now come to the induction part.

Induction: Suppose that (11) holds for some k. Let $A = \{pole^i_m(T_{k'}) \mid T_{k+1}\; isa\; T_{k'},\; 1 \leq k' \leq k\}$ and $B = \{T' \in Pole^i_m \mid T_{k+1} \prec T'\}$.

- $A \subset B$: Let $T' \in A$. $T' \in Pole^i_m$ and $\exists k'$ such that $T_{k+1} \prec T_{k'} \preceq T'$. Thus, $T' \in B$.
- $min_{\preceq}B \subset A$: Let $T' \in min_{\preceq}B$. As $T_{k+1} \prec T'$ either (i) $T_{k+1}\; isa\; T'$ or (ii) there exists an ascending chain C of the partially ordered set (Θ, isa) starting from T_{k+1} and finishing at T'.

  case (i): by (8), $pole^i_m(T') = T'$. Hence, $T' \in A$.
  case (ii): Let $C = T_{k+1}\, isa\, T^0 \ldots isa\, T'$, we first prove that $closest\text{-}poles^i_m(T^0) = \{T'\}$.

  - $T' \in closest\text{-}poles^i_m(T^0)$: Suppose there exists $T'' \in Pole^i_m$ such that $T^0 \preceq T'' \prec T'$, then $T^{k+1} \preceq T'' \prec T'$, which contradicts that $T' \in min_{\preceq}B$.
  - $closest\text{-}poles^i_m(T^0) \subseteq \{T'\}$: Suppose there exists $T'' \in closest\text{-}poles^i_m(T^0)$ such that $T' \neq T''$. Then, by Theorem 3, $pole^i_m(T^0) = T^0$. As $T_{k+1} \prec T^0 \prec T'$, $T' \notin closest\text{-}poles^i_m(T_{k+1})$, which contradicts that $T' \in min_{\preceq}B$.

Thus $pole^i_m(T^0) = T'$, and since $T_{k+1} \preceq T^0$, $T' \in A$.

We can now apply the above claim which gives that $min_{\preceq}\{pole^i_m(T_k) \mid T_{k+1}\ isa\ T_{k'},$ $1 \leq k' \leq k\} = min_{\preceq}\{T' \in Pole^i_m \mid T_{k+1} \prec T'\} = closest\text{-}poles^i_m(T_{k+1})$. Thus (11) holds for $k+1$.

□

Note that the poles of the direct supertypes of a type T are also supertypes of T, thus this verification is not needed in the computation of the closest poles of T.

Nonetheless, (11) still requires to compute the minimum over a set, and $min_{\preceq}E$ has a cost of $|E|^2$. By Theorem 3, the Pole Computation Algorithm only needs to know if $|closest\text{-}poles^i_m(T)|$ is 0, 1, or more, and if there is a single closest pole, we need to compute it. In this latter case, the closest pole of T has the greatest rank in $(\Theta, \leq)$ among all the poles of T.

Based on these considerations, we propose to compute the *pseudo-closest* poles of T. We assume that pole types are ranked in the ascending order of $\leq$. The algorithm consists of three iterations over the set $\{pole^i_m(T') \mid T\ isa\ T'\}$. In the first iteration, this set is assigned to the variable *candidates*. In the next iteration, the type T^c with the greatest rank is returned, and finally it is checked if it is a subtype of all the other poles. If there is one pole T' for which T^c is not a subtype then T' is also returned and the computation stops.

The algorithm assumes that each type has a *pole-rank* attribute used for pole types only. Because poles are found in the ascending order of $\leq$, pole ranks are also generated in this ascending order. This algorithm is used instead of *closest-poles*.

5.3 Testing the Subtyping Relationship

In this section, we show how to optimize the many subtyping tests that are needed in the *pseudo-closest-poles* algorithm.

We associate with each pole the set of poles of its direct supertypes, implemented as a vector of bits. Because this number of poles is usually small, a few bit vectors are needed.

input : a type T, an argument position i, a generic function m
output : a set of poles *pseudo-closest*

Iteration 1: compute *candidates*
$candidates \leftarrow \{pole^i_m(T') \mid T \ isa \ T'\}$

if $candidates = \emptyset$ then
  $pseudo\text{-}closests = \emptyset$
else
  Iteration 2: search for type T^c with greatest rank
  $T^c \leftarrow$ any-element($candidates$)
  for T' in $candidates$
    if $T'.pole\text{-}rank > T^c.pole\text{-}rank$ then
      $T^c \leftarrow T'$

  Iteration 3: check T^c is a subtype of all members of *candidates*
  $pseudo\text{-}closest \leftarrow \{T^c\}$
  for T' in $candidates$
    if $T' \not\preceq T^c$ then
      insert T' into *pseudo-closest*
      break

Figure 8: Pseudo-Closest Poles Algorithm

The vector is built during the computation of poles and is released when all poles in a given dimension have been obtained. The subtyping test can then be performed in constant-time by testing the value of a bit.

The poles in a given dimension form an ascending chain of $(\Theta, \leq)$, denoted $(T^k)_{1 \leq k \leq l}$, where k indicates the rank of the pole in the chain.

Definition 5.1 *For each k in $\{1, \ldots, l\}$, the bit vector $\mathcal{H}^k$ of length l associated with T^k is defined as:* $\forall k',\ 1 \leq k' \leq l,\quad \mathcal{H}^k(k') = 1 \quad \Leftrightarrow \quad T^{k'} \succ T^k$

$$\mathcal{H}^k(k') = 0 \quad \Leftrightarrow \quad T^{k'} \not\succ T^k$$

The following theorem allows to build these arrays at a very low cost:

Theorem 5 *For each k in $\{1, \ldots, l\}$, we have:*

$$\mathcal{H}^k = \bigvee_{T^{k'} \in \{pole^i_m(T') \ \mid \ T^k \ isa \ T'\}} \mathcal{H}^{k'} \vee (2^{k'}) \tag{14}$$

The symbol $\vee$ denotes the logical "OR" operation, and $2^{k'}$ is the bit vector with a "1" at position k' and "0" everywhere else.

Proof: As $\mathcal{H}^k$ is a bit vector, (14) is equivalent to:

$$\mathcal{H}^k[k'] = 1 \Leftrightarrow (\ \exists k'' \text{ such that } (\quad T^{k''} \in \{pole^i_m(T') \mid T^k\ isa\ T'\} \text{ and } (\ \mathcal{H}^{k''}[k'] = 1 \text{ or } k' = k'')))$$

Using Definition 5.1, this is equivalent to:

$$\begin{aligned} T^k \prec T^{k'} \quad &\Leftrightarrow \quad (\ \exists T^{k''} \in \{pole^i_m(T') \mid T^k\ isa\ T'\} \text{ such that } (\ T^{k''} \prec T^{k'} \text{ or } T^{k''} = T^{k'}\ )) \\ &\Leftrightarrow \quad (\ \exists T^{k''} \in \{pole^i_m(T') \mid T^k\ isa\ T'\} \text{ such that } T^{k''} \preceq T^{k'}\ ) \end{aligned}$$

We prove now the last equivalence.

$\Leftarrow$: As $T^k \prec T' \preceq T^{k''}$, and $T^{k''} \preceq T^{k'}$, by transitivity of $\preceq$, $T^k \prec T^{k'}$.

$\Rightarrow$: As $T^{k'} \in Pole^i_m$ and $T^k \prec T^{k'}$, by Definition 4.6, $\exists T^{k''} \in closest\text{-}poles^i_m(T^k)$ such that $T^{k''} \preceq T^{k'}$. Let $A = \{pole^i_m(T') \mid T^k\ isa\ T'\}$. From Theorem 4, $closest\text{-}poles^i_m(T^k) = min_{\preceq}A$. Hence $T^{k''} \in min_{\preceq}A$, and $T^{k''} \in A$.

□

This result enables to easily build a vector $\mathcal{H}^k$ as the logical OR between the bit vectors associated with the poles of the direct supertypes of T^k and a bit vector in which 1-valued bits refer to the direct supertypes of T^k.

The simple test of Figure 9 tests if a pole T^k is a subtype of a pole $T^{k'}$. It assumes that $\mathcal{H}$ is implemented as an array of arrays of bits, such that $\mathcal{H}^k$ is the k-th array of bits, and $\mathcal{H}^k[k']$ is the k'-th bit in this array.

```
input   : two poles T^k and T^k'
output  : true iff T^k ≺ T^k', else false
if T^k.pole-rank > T^k'.pole-rank then
    if H^k[k'] = 1 then
        return(true)
return(false)
```

Figure 9: Subtyping Test

The ranking of poles in $(\Theta, \leq)$, and the construction of bit vectors, are progressively done as long as a new pole is found by the routine in Figure 10, which must be invoked by the algorithm of Figure 6.

As a pole T^k cannot be a subtype of $T^{k'}$ if $k < k'$, we have $\forall k, k', (k < k' \Rightarrow \mathcal{H}^k(k') = 0)$. Hence $\mathcal{H}^k$ can have an effective length of $k-1$, which allows to allocate this vector as soon as pole T^k is found.

input : A new pole T, its closest poles *closest-poles*
output : modified T, modified $\mathcal{H}$, modified *last-pole-rank*

increment *last-pole-rank*
$T.pole\text{-}rank \leftarrow last\text{-}pole\text{-}rank$
for T_c in $\{pole^i_m(T') \mid T \ isa \ T'\}$ do
$\quad \mathcal{H}^{T.pole\text{-}rank} \leftarrow \mathcal{H}^{T.pole\text{-}rank} \vee \mathcal{H}^{T_c.pole\text{-}rank} \vee 2^{T_c.pole\text{-}rank}$

Figure 10: Pole Initialization Routine

5.4 Worst-case complexity

In this section, we note $|m|$ the number of methods of a generic function m, $\mathcal{E}$ the number of edges in the type graph, and we use the following definition:

Definition 5.2 *Let $T \in \Theta$, the set of direct supertypes of T is noted $dst(T)$:*

$$dst(T) = \{T' \mid T \ isa \ T'\}$$

Cost Units

Figure 11 summarizes the cost units of the basic operations used in our algorithms. All these operations are supposed to execute in constant time. Bit vectors have a fixed length hence they can be represented as an array of integers, hence $BitTest$ and $BitOr$ execute in constant time. For sets, we do not assume any duplicate removal, hence *InsSet* is constant. We also assume that the number of elements is memorized into the set representation, hence *SetCard* executes in constant time.

Unit	Description of Basic Operation
IntComp	comparison between two integers
VarAssg	variable assignment
BitTest	test the value of an entry in an array of bits
BitOr	logical "OR" between two arrays of bits
InsSet	insertion of an element into a set
SetCard	yields the cardinality of a set
SetElem	take an element out of set

Figure 11: Cost Units

Subtyping Test

The cost of the subtyping test in Figure 9 is:

$$SbTest = IntComp + BitTest$$

Pseudo Closest Poles

The *pseudo-closest-poles* algorithm in Figure 8 essentially consists of three loops over the set of direct supertypes of a type T.

- Loop 1: collects in *candidates* the poles of the direct supertypes. This costs

$$L_1(T) = |dst(T)| \times InsSet$$

- Loop 2: searches for the pole T^c with the greatest rank. Each iteration consists at worst of a comparison between integers and an assignment. This costs

$$L_2(T) = |dst(T)| \times (IntComp + VarAssg)$$

- Loop 3: checks that T^c is a subtype of all the other poles. It includes the comparison $T' \not\preceq T^c$ which consists of an equality comparison between the ranks of T' and T^c, and a subtyping test. Hence each iteration consists at worst of an equality comparison, a subtyping test, and an insertion into *pseudo-closest*. This costs

$$\begin{aligned} L_3(T) &= |dst(T)| \times (IntComp + SbTest + InsSet) \\ &= |dst(T)| \times (2 \times IntComp + BitTest + InsSet) \end{aligned}$$

To sum up, this algorithm costs

$$PCP(T) = L_1(T) + L_2(T) + L_3(T) = |dst(T)| \times (3 \times IntComp + BitTest + 2 \times InsSet)$$

Pole Initialization

The pole initialization routine in Figure 10 essentially consists of a loop over the direct supertypes. Its cost is:

$$PI(T) = |dst(T)| \times (2 \times BitOr + VarAssg)$$

```
input   : an ordered set of types (Θ, ≤), a generic function m, an argument position i
output  : a set of poles Poles, modified types
Poles ← ∅
Step 1: mark primary poles
increment current-mark
for m_k in m.methods do
    m_k.formals[i].mark ← current-mark
Step 2:
for T in (Θ, ≤) do
    if T.mark = current-mark then   // tests if T ∈ Static^i_m (case 3a of Theorem 3)
        pole-init(T)                // T is a primary pole
        T.pole ← T
        insert T into Poles
    else
        closest ← pseudo-closest-poles(T, m, i)
        size-closest ← |closest|
        if size-closest = 0 then    // case 1 of Theorem 3
            T.pole ← 0
        else
            if size-closest = 1 then    // case 2 of Theorem 3
                T.pole ← first(closest)
            else                        // case 3b of Theorem 3
                pole-init(T)            // T is a secondary pole
                T.pole ← T
                insert T into Poles
```

Figure 12: Detailed Pole Computation Algorithm

Pole Computation

We compute the worst-case complexity of pole computation on the detailed algorithm of Figure 12. In particular, the membership test of a type T in $Static^i_m$ can be done immediately by marking each type of $Static^i_m$. An integer mark is generated before each pole computation and assigned to the variable *current-mark*. This new mark is used to identify the elements of $Static^i_m$, by assigning it to the attribute $mark$ of each type in $Static^i_m$. Then, to test if a type T is a member of $Static^i_m$, it suffices to test if $T.mark$ is equal to *current-mark*. We use a record-like representation of generic functions and methods. Generic functions have an attribute *methods* that represents the set of methods, and methods have an attribute $formals$ that represents the array of their formal arguments, which are primary poles. The algorithm essentially consists of two loops:

- Loop 1 (lines 3-4): this loops marks the ith formal argument of each method of the generic function m, which costs

$$L_1 = |m| \times VarAssg$$

- Loop 2 (lines 5-21): this is the main loop over Θ, which consists of several tests.
  - If T is a primary pole, only lines 6-9 are executed, which costs

$$L_2^1(T) = IntComp + PI(T) + VarAssg + InsSet$$

  - If T has no pole, lines 6 and 11-14 are executed, which costs

$$L_2^2(T) = 2 \times IntComp + PCP(T) + 3 \times VarAssg + SetCard$$

  - If $pole^i_m(T) = T', T \neq T'$, lines 6, 11-13, and 16-17 are executed, which costs

$$L_2^3(T) = 3 \times IntComp + PCP(T) + 3 \times VarAssg + SetCard + SetElem$$

  - If T is a secondary pole, lines 6, 11-13, 16, and 19-21 are executed, which costs

$$L_2^4(T) = 3 \times IntComp + PCP(T) + 3 \times VarAssg + SetCard + PI(T) + InsSet$$

In the worst case, the cost is $max\{L_2^1(T), L_2^2(T), L_2^3(T), L_2^4(T)\}$, which is bounded by

$$L_2(T) = 3 \times IntComp + PCP(T) + PI(T) + 3 \times VarAssg + SetCard + InsSet + SetElem$$

The total cost of pole computation is

$$\begin{aligned}
PC &= L_1 + \sum_{T \in \Theta} L_2(T) \\
&= |m| \times VarAssg + \sum_{T \in \Theta} (3 \times IntComp + PCP(T) + PI(T) + 3 \times VarAssg + SetCard \\
&\qquad + InsSet + SetElem) \\
&= |m| \times VarAssg + \sum_{T \in \Theta} (3 \times IntComp + 3 \times VarAssg + SetCard + InsSet + SetElem \\
&\qquad + |dst(T)| \times (3 \times IntComp + BitTest + 2 \times InsSet2 \times BitOr \\
&\qquad\qquad + VarAssg)) \\
&= |m| \times VarAssg + |\Theta| \times (3 \times IntComp + 3 \times VarAssg + SetCard + InsSet + SetElem) \\
&\quad + (\sum_{T \in \Theta} dst(T)) \times (3 \times IntComp + BitTest + 2 \times InsSet2 \times BitOr + VarAssg)
\end{aligned}$$

As $\sum_{T \in \Theta} dst(T) = \mathcal{E}$, we finally have

$$PC = \mathcal{O}(|m| + |\Theta| + \mathcal{E})$$

6 Filling Up Dispatch Tables

The computation of poles determines the structure of the dispatch table and the associated argument-arrays. Filling up the dispatch table requires to compute the MSA method for each signature of poles. Thus, the more the table is compressed, the less MSA computations are needed. In this section, we optimize the computation of a single MSA method. We first show that the MSA method can be computed on a subset of the applicable methods obtained by examining the signatures of poles in an order compatible with signature precedence. We explain how such an order is obtained. We finally give the algorithm to fill up the dispatch table.

6.1 MSA Method Computation

We first define the notions of pole signature and candidate signature.

Definition 6.1 *Given a generic function m of arity n, the set of pole signatures is:*

$$Poles_m = \prod_{i=1}^{n} Pole^i_m$$

Definition 6.2 *For each $(T^1, \ldots, T^n)$ in $Dynamic_m$, the set of candidate signatures of $(T^1, \ldots, T^n)$ is:*

$$\begin{aligned} &candidate\text{-}signatures_m((T^1, \ldots, T^n)) = \\ &\quad \{(T'_1, \ldots, T'_n) \ \mid \ \exists i \in \{1, \ldots, n\}, \exists T' \in \Theta \ s.t. \ T'_i = pole^i_m(T') \ and \ T_i \ isa \ T \\ &\qquad and \forall j \neq i, \ T'_j = T_i\}. \end{aligned}$$

Example 6.1: Consider the types and methods in Figure 2. The 1-poles are A, B and D, and the 2-poles are B, C and D. The candidate signatures of (D, D) are obtained by substitution with the poles of the direct supertypes. The 1-poles of D's direct supertypes are A and B, and their 2-poles is simply B. Hence the candidate signatures of (D, D) are $(A, D), (B, D)$, and (D, B). □

Theorem 6 *For each $s \in Poles_m$, which is not the signature of a method, we have:*

$$MSA(m(s)) = min_{\preceq_s}\{MSA(m(s')) \mid s' \in candidate\text{-}signatures_m(s)\}$$

Proof: Let $s = (T^1, \ldots, T^n)$. As $\{MSA(m(s))\} = min_{\preceq}\{m_k \mid m_k \succeq s\}$, we note $A = \{MSA(m(s')) \mid s' \in candidate\text{-}signatures_m(s)\}$ and $B = \{m_k \mid m_k \succeq s\}$.

- $A \subset B$: let $m_k \in A$, and $s' \in candidate\text{-}signatures_m(s)$ s.t. $m_k = MSA(m(s'))$. From Definition 6.2, $s \preceq s'$. From Definition 2.1, m_k is transitively applicable to s. By definition of B, $m_k \in B$.
- $min_{\preceq}B \subset A$: we proceed by contradiction. Let $m_s \in min_{\preceq}B$, i.e. $m_s = MSA(m(s))$. We assume that $m_s \notin A$. Let $T^1_s, \ldots, T^n_s$ be the formals of m_s, we have $MSA(m(T^1_s, \ldots, T^n_s)) = m_s$. As $m_s \notin A$, we have $(T^1_s, \ldots, T^n_s) \notin candidate\text{-}signatures_m(s)$. By

Definition 2.1, and as s is not the signature of a method, $s \prec (T_s^1, \ldots, T_s^n)$. We build a candidate signature s_c such that $s_c \prec (T_s^1, \ldots, T_s^n)$. Two cases may occur:

- $\exists! i \in \{1, \ldots, n\}$ such that $T_s^i \succ T^i$. As $T_s^i \in Pole_m^i$, by (8) $pole_m^i(T_s^i) = T_s^i$. We show by contradiction that T_s^i is not a direct supertype of T_i: assuming T^i *isa* T_s^i, we have $(T_s^1, \ldots, T_s^n) \in candidate\text{-}signature_m(s)$, a contradiction. Hence $\exists T_0 \in \Theta$ s.t. T^i *isa* $T_0 \prec T_s^i$. As $T_s^i \in Static_m^i$, by Definition 4.4, $T_0 \in Dynamic_m^i$. Let $T'_0 = pole_m^i(T_0)$. As $T_s^i \in Pole_m^i$ and $T_0 \prec T_s^i$, by Corollary 4.2, $T'_0 \preceq T_s^i$. As T^i *isa* T_0, $T'_0 = pole_m^i(T_0)$ and $(T_s^1, \ldots, T_s^n) \notin candidate\text{-}signatures_m(s)$, by Definition 6.2, $T'_0 \neq T_s^i$. Hence $s_c = (T^1, \ldots, T^{i-1}, T'_0, \ldots, T_n)$ is a candidate signature and $s_c \prec (T_s^1, \ldots, T_s^n)$.
- $\exists i, j \in \{1, \ldots, n\}$, $i \neq j$, s.t. $T_s^i \succ T^i$ and $T_s^j \succ T^j$. Let T_0 such that T_i *isa* $T_0 \preceq T_s^i$. As $T_s^i \in Static_m^i$, by Definition 4.4, $T_0 \in Dynamic_m^i$. Let $T'_0 = pole_m^i(T_0)$. As $T_s^i \in Pole_m^i$ and $T_0 \prec T_s^i$, by Corollary 4.2, $T'_0 \preceq T_s^i$. Let s_c be the candidate signature $(T^1, \ldots, T^{i-1}, T_0, \ldots, T^n)$. As $T_j \prec T_s^j$, $s_c \prec (T_s^1, \ldots, T_s^n)$.

As $s_c \prec (T_s^1, \ldots, T_s^n)$, be Definition 2.1, m_s is applicable to s_c. As $m_s \notin A$ and s_c is a candidate signature, $m_s \neq MSA(m(s_c))$. As $s \prec s_c$, by Definition 2.3, m_s cannot be the MSA method for $m(s)$, a contradiction. Hence $m_s \in A$.

As the set of methods applicable to $m(s)$, ordered by $\preceq_s$, is a partially ordered set, by (12), we have $min_{\preceq_s} A = min_{\preceq_s} B$. □

This theorem is useful if the MSA methods of the candidate signatures are known before computing $MSA(m(s))$. We propose to ensure this property by considering signatures in the ascending order of $\leq$, $\leq$ being the opposite of a linear extension of the precedence order $\preceq$:

$$(\forall s, s' \in \Theta^n, (s \leq s' \text{ or } s' \leq s)) \text{ and } (\forall s, s' \in \Theta^n, (s \preceq s' \Rightarrow s' \leq s))$$

As for pole computation, this order of pole signatures ensures that for each signature s, the signatures s', such that $s \prec s'$, are considered before s (since $s' < s$). As all the candidate signatures of s are in this case, their MSA methods are computed before considering s, and Theorem 6 can be applied. Note that the same order is used in the method disambiguation algorithm presented in [AD96], which allows to perform table fill-up and method disambiguation at the same time.

6.2 Ordering the Pole Signatures

Ordering the pole signatures in an order that is compatible with precedence comes down to transform a partially ordered set $(\mathcal{S}, \preceq)$ into a totally ordered set $(\mathcal{S}', \leq)$, where $\mathcal{S}$ is the set of pole signatures. A classical algorithm is given in [Knu73]. The basic idea is to pick a first element that has no predecessor, remove this element from $\mathcal{S}$, append it to the originally empty set $\mathcal{S}'$, and start over until $\mathcal{S}$ is empty. In the case of pole signatures, it is necessary to scan the set of pole signatures to find that a given signature has no predecessor. Hence, ordering the pole signatures has an a priori complexity of $O(|Poles_m|^2)$.

However, it is possible to obtain a complexity of $O(|Poles_m|)$ if the poles of each dimension, $Pole^i_m$, are themselves sorted in an order compatible with signature precedence. Indeed, it suffices to produce the signatures in the lexicographic ordering generated by the total orders on the poles. This ordering is inexpensive as poles are already produced in this order by the algorithm of Figure 6. The total order $\leq$ on Θ^n is defined as follows:

Definition 6.3 *Given a generic function m of arity n, and $(T_1, \ldots, T_n), (T'_1, \ldots, T'_n) \in Poles_m$:*
$$((T_1, \ldots, T_n) \leq (T'_1, \ldots, T'_n)) \quad \Leftrightarrow \quad (\ (T_1, \ldots, T_n) = (T'_1, \ldots, T'_n) \ \textit{or}$$
$$(\exists i_0,\ i_0 = min\{i \mid T_i \neq T'_i\} \ \textit{and} \ T_{i_0} < T'_{i_0}))$$

Theorem 7 *Given a generic function m, the relation $\leq$ defines a total order on $Poles_m$ which is an extension of the precedence order $\preceq$.*

Proof: Let n be the arity of m. Let $s = (T_1, \ldots, T_n), s' = (T'_1, \ldots, T'_n) \in Poles_m$.

- Antisymmetry: We assume that $s \leq s'$ and $s' \leq s$. We proceed by contradiction: assuming $s \neq s'$, let $i_0 = min\{i \mid T_i \neq T'_i\}$. By Definition 6.3, $s \leq s'$ implies $T_{i_0} < T'_{i_0}$, and $s' \leq s$ implies $T'_{i_0} < T_{i_0}$, a contradiction.
- Transitivity: Let $s'' = (T''_1, \ldots, T''_n) \in Poles_m$. We assume that $s \leq s'$ and $s' \leq s''$. We consider two cases:

- $s = s'$ or $s' = s''$: by substitution of s' with s or with s'', $s \leq s''$.
- $s \neq s'$ and $s' \neq s''$: let $i_0 = min\{i \mid T_i \neq T'_i\}$, $i'_0 = min\{i \mid T'_i \neq T''_i\}$, and $i''_0 = min\{i_0, i'_0\}$. Then $\forall i, i < i''_0$, $T_i = T''_i$. If $i''_0 = i_0$ and $i''_0 < i'_0$, then $T_{i''_0} < T'_{i''_0}$ and $T'_{i''_0} = T''_{i''_0}$, hence $T_{i''_0} < T''_{i''_0}$. In the same way, if $i''_0 = i_0$ and $i''_0 < i'_0$, then $T_{i''_0} < T''_{i''_0}$. Finally if $i''_0 = i_0$ and $i''_0 = i'_0$, then $T_{i''_0} < T'_{i''_0} < T''_{i''_0}$. By Definition 6.3, this implies $s \leq s''$.

- Total order: If $s = s'$ then $s \leq s'$. If not, $\{i \mid T_1 \neq T'_i\} \neq \emptyset$, hence $i_0 = min\{i \mid T_i \neq T'_i\}$ is defined. As $\leq$ is a total order over Θ, either $T_i \leq T'_i$ or $T'_i \leq T_i$, and respectively $s \leq s'$ or $s' \leq s$.
- Extension of $\preceq$: We assume that $s \preceq s'$. If $s = s'$ then $s \leq s'$. If not, let $i_0 = min\{i \mid T_i \neq T'_i\}$. As $s \preceq s'$, by definition of $\preceq$, $T_{i_0} \preceq T'_{i_0}$. As $T_{i_0} \neq T'_{i_0}$, $T_{i_0} \prec T'_{i_0}$. As $<$ is an extension of $\leq$ on Θ, $T_{i_0} < T'_{i_0}$. Hence by Definition 6.3, $s \preceq s'$.

□

Example 6.2: The table in Figure 13 represents the pole signatures of the methods and types of Figure 2. The order on 1-poles (resp. 2-poles) in lines (resp. columns) is compatible with argument subtype precedence. A total order of $Poles_m$ is a path through this table. Such a path is compatible with signature precedence if it traverses each signature s before the signatures on the right and below s. The path given by a lexicographic ordering, as shown on Figure 13, satisfies the condition. For example, the signatures that are more specific than (B, C) are all included in the grey area. □

6.3 Table Fill Up Algorithm

Figure 14 gives the fill up algorithm of table *dispatch* for a generic function m. This algorithm proceeds in three steps. The first step invokes the function *OrderedPoleSignature* that builds the ordered poles signatures $(Poles_m, \preceq)$, using the array of pole lists P, where $P[i]$ is $(Pole^i_m, \leq)$. The second step fills the cells of *dispatch* which indices are signatures of methods. The value of each of these cells is simply the corresponding method. The third step considers the other pole signatures ordered by $\leq$ using Theorem 6. It collects the MSA

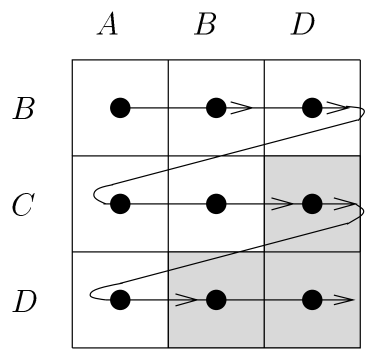


Figure 13: Order of Pole Signatures

input : a generic function m, an array of pole lists P, an empty table $dispatch$
side-effect: filled $dispatch$ table

Step 1: initialization
$(Poles_m, \leq) \leftarrow OrderedPoleSignatures(P)$

Step 2: dispatch of method signatures
for m_k in $m.methods$ do
  $dispatch[m_k.formals] \leftarrow m_k$

Step 3: computation of the MSA methods
for s in $(Poles_m, \leq)$ do
  if $dispatch[s] = 0$ then // s is not a method signature
    $candidates \leftarrow \{dispatch[s'] \mid s' \in candidate\text{-}signatures(s)\}$
    if $candidates \neq \emptyset$ then
      $dispatch[s] \leftarrow most\text{-}specific(candidates)$

Figure 14: Table Fill Up Algorithm

method of candidate signatures into a variable *candidates*, and assigns the most specific of these signatures to the cell associated with the current signature in *dispatch*.

6.4 Most Specific Method Computation

To compute the complexity of the Table Fill Up Algorithm, we need to specify the *most-specific* function used to compare methods. This function depends on the method specificity criterium of the language. In this section, for simplicity, we use argument subtype precedence, without any additional criterium. This ordering is monotonous as the same ordering of methods applies for every invocation signature.

In this case,the specificity relationship on methods is the same as the relation $\preceq$ on their signatures. Hence the order $\leq$ on pole signatures also defines a linear extension of the specificity order for methods. Given a generic function m, we rank all methods of m in the ascending order of their signatures in $(Poles_m, \leq)$, the rank of method $m_k(T_k^1, \ldots, T_k^n)$ being k. As in Section 5.3, with each method m_k we associate a bit-vector called higher-methods and noted $higher\text{-}methods(m_k)$. The k'th bit of $higher\text{-}methods(m_k)$ is 1 iff m_k is more specific than $m_{k'}$. As shown by Theorem 5 in the context of higher-poles's arrays, $higher\text{-}methods(m_k)$ are computed as a combination of the methods higher than the MSA for the candidate signatures of $(T_k^1, \ldots, T_k^n)$.

Operationally, the ranking of methods and the construction of the higher methods vectors are done during the traversal of $(Poles_m, \leq)$ required by the Table Fill Up Algorithm. In this traversal, the signatures of methods are identified by existing table's cells. The corresponding methods are then treated by the Higher Methods Initialization routine in Figure 15. This routine assumes that both the rank and the higher methods are represented as attributes $rank$ and $higher\text{-}methods$. This solution allows to treat the methods in the ascending order of $\leq$. Thus, the higher methods of the MSA methods of the candidate signatures are already defined.

input : a method $method$, a table $dispatch$
side-effect: modified $method$, modified $last\text{-}method\text{-}rank$

1. $method.rank \leftarrow last\text{-}method\text{-}rank$
2. increment $last\text{-}method\text{-}rank$
3. for other in $\{dispatch[s'] \mid s' \in candidate\text{-}signatures(method.formals)\}$ do
4. $\quad method.higher\text{-}methods \leftarrow method.higher\text{-}methods \vee other.higher\text{-}methods \vee 2^{other.rank}$

Figure 15: Higher Methods's Initialization Routine

The computation of the most specific method using the higher methods bits, in Figure 16, proceeds in three steps. The first step collects the MSA method of candidate signatures in $candidates$. In the second step, as $\leq$ is an extension of $\preceq$, the most specific method is searched as the method m^c with the highest rank. The third step detects method selection ambiguities, by verifying that m^c is more specific than the other methods, using the higher methods bits.

input : a signature s, a table $dispatch$
output : a most specific method m^c, "ambiguous" in case of ambiguity

Step 1: compute $candidates$
$candidates \leftarrow \{dispatch[s'] \mid s' \in candidate\text{-}signatures(s)\}$
Step 2: search method with highest rank
$m^c \leftarrow$ any-element$(candidates)$
for m in $candidates$ do
    if $m^c.rank < m.rank$ then
        $m^c \leftarrow m$

Step 3: check for ambiguity
for m in $candidates$ do
    if $m^c.higher\text{-}methods[m.rank] \neq 1$ then
        $m^c \leftarrow$ "ambiguous"
        break

Figure 16: Most Specific Algorithm

6.5 Worst-Case Complexity

We use the cost units introduced in Figure 11.

Table Access

The cost of an access in a dispatch table of dimension n is proportional to n:

$$TabAcc(n) = n \times TabAcc(1)$$

Most Specific Method

This routine consists of three iterations over the candidate signatures. It is similar to the pseudo-closest pole computation, and costs at most

$$MSM = n \times K \times (InsSet + IntComp + 2 \times VarAssg + BitTest)$$

Higher Methods Initialization Routine

This routine essentially consists of an iteration over the MSA methods of candidate signatures. Let K be the maximum number of direct supertypes of all types. The number of candidate signatures is less than $n \times K$. The routine costs

$$HMI = n \times K \times (2 \times BitOr + VarAssg)$$

Table Fill Up

The algorithm in Figure 14 essentially consists of three iterations.

- Iteration 1 (line 1): builds the list of pole signatures, which costs

$$I_1 = |Poles_m| \times InsSet$$

- Iteration 2 (lines 2-3): assigns the cells of the signatures of methods, which costs

$$I_2 = |m| \times (TabAcc(n) + VarAssg)$$

- Iteration 3 (lines 4-8): assigns the other cells.
  - The detection of the signatures of methods costs

$$I_3^1 = |Poles_m| \times (TabAcc(n) + IntComp)$$

  - The computation of the MSA methods of candidate signatures costs at each step

$$I_3^2 = n \times K \times TabAcc(n)$$

  - Testing if *candidates* is empty amounts to testing if $|candidates| = 0$, which costs at each step

$$I_3^3 = SetCard + IntComp$$

  - The assignment in line 8 costs

$$I_3^4 = TabAcc(n) + VarAssg + MSM$$

As I_3^2 to I_3^4 do not apply to method signatures, and taking into account the initialization of the higher-methods bits, this iteration costs

$$\begin{aligned}
I_3 &= I_3^1 + (|Poles_m| - |m|) \times (I_3^1 + I_3^3 + I_3^4) + |m| \times HMI \\
&= |Poles_m| \times (TabAcc(n) + IntComp) \\
&\quad +(|Poles_m| - |m|) \times (n \times K \times TabAcc(n) + SetCard + IntComp \\
&\qquad\qquad + TabAcc(n) + VarAssg + MSM) \\
&\quad +|m| \times n \times K \times (2 \times BitOr + VarAssg) \\
&= |Poles_m| \times (2 \times TabAcc(n) + 2 \times IntComp + SetCard + VarAssg) \\
&\quad +|Poles_m| \times n \times K \times (TabAcc(n) + InsSet + IntCompt + 2 \times VarAssg \\
&\qquad\qquad + BitTest) \\
&\quad -|m| \times (SetCard + IntComp + TabAcc(n) + VarAssg) \\
&\quad +|m| \times n \times K \times (2 \times BitOr - TabAcc(n) - InsSet - IntCompt \\
&\qquad\qquad - VarAssg - BitTest)
\end{aligned}$$

To sum up, table fill up costs

$$\begin{aligned}
TFU &= I_1 + I_2 + I_3 \\
&= |Poles_m| \times (InsSet + 2 \times TabAcc(n) + 2 \times IntComp + SetCard + VarAssg) \\
&\quad +|Poles_m| \times n \times K \times (TabAcc(n) + InsSet + IntCompt + 2 \times VarAssg + BitTest) \\
&\quad +|m| \times (TabAcc(n) + VarAssg - SetCard - IntComp - TabAcc(n) - VarAssg) \\
&\quad +|m| \times n \times K \times (2 \times BitOr - TabAcc(n) - InsSet - IntCompt - VarAssg - BitTest) \\
&= |Poles_m| \times (InsSet + 2 \times n \times TabAcc(1) + 2 \times IntComp + SetCard + VarAssg) \\
&\quad +|Poles_m| \times n \times K \times (n \times TabAcc(1) + InsSet + IntCompt + 2 \times VarAssg + BitTest) \\
&\quad -|m| \times (SetCard + IntComp) \\
&\quad -|m| \times n \times K \times (-2 \times BitOr + n \times TabAcc(1) + InsSet + IntCompt + VarAssg \\
&\qquad + BitTest)
\end{aligned}$$

Assuming the last term is negative, and as $|Poles_m| \leq |\Theta|^n$, we obtain

$$TFU = \mathcal{O}(|Poles_m| \times (1+n) \times (1 + n \times K)) = \mathcal{O}(|\Theta|^n \times n^2 \times K))$$

7 Implementation and Measurements

This section focuses on the run-time dispatching using compressed dispatch tables. We first describe the representation of the run-time data structures. In particular, we optimize the size of argument-arrays using bytes or 16-bits words. We also describe the dispatch code since its size must also be counted. We then describe the global algorithm to build the dispatch structures with group argument-arrays using bytes or 16-bits words. Finally, we present an application with multi-methods, and our experimental results.

7.1 Run-Time Structures and Dispatch Code

The data structures used at run-time are slightly different from those used at compile time. In the following, we consider an n-ary generic function m. We note ω the size in bytes of a memory word (in practice, $\omega = 4$ or 8).

Since run-time dispatch only needs the address of the code to execute, we assume that at run-time, each dispatch table's cell holds the address of a method's code.

We assume that these addresses are stored contiguously, starting from a base address β. We also assume that a n-dimensional table is stored as a nesting of one-dimensional arrays. The top-level array is associated with the first argument position, and the low-level arrays are associated with the last argument position. For each dimension, the associated argument-array stores the offset of the cells with respect to the base of a one-dimensional array. The function m_arg_i (introduced in Section 3.3) associates each type with the corresponding offset in the i-th dimension. This offset is the multiplication of the pole number of the type by a constant γ_i^m, that depends on the number of poles for the next argument positions. Hence given an invocation signature $m(o_1, \ldots, o_n)$, the address of the code to execute appears at address $\beta + \sum_{i=1}^{n} m\_arg_i[o_i.type\_index]$. Depending on the value of the largest offset in an argument-array, its content can be stored on 8-bits, 16-bits or 32-bits memory words[1]. In practice, 32-bits words were never needed in our measures, hence we only assume 8- and 16-bits words in the following.

We assume that the operations necessary to perform the dispatch are replicated for each generic function. This allows to have the base addresses of the argument-arrays and table as constants in the code.

For each invocation, the application's object code includes the parameter passing code, and a call to the dispatch routine. This is equivalent to a static invocation and cannot be counted as method selection overhead. Then the dispatch routine finds the address of the code to execute, based on the relevant arguments, and jumps to this code. Finally this code performs the necessary stack adjustments. Again, these adjustments cannot be counted as method selection overhead.

In the case of a SPARC processor, the dynamic dispatch code for n target arguments needs $5n + 3$ operations. We give in Figure 17 the dispatch code[2] for a generic function with two target arguments, without type checking. It assumes that both argument-arrays are made of bytes, and that the arguments of the invocation are referred to by registers `i0` and `i1`. Apart from giving the space overhead incurred by multiple dispatching, this code also obviates that run-time dispatch is done efficiently and in constant time, the effective time depending on the version of the processor.

7.2 Implementation of Group Argument-Arrays

We use the same coloring algorithm as [DMSV89] to construct group argument-arrays. Each selector is associated with the set of types to which it is applicable, i.e. to which one of its formal arguments is applicable. The conflict set of a selector is the set of the other selectors which set of applicable types intersect with its own set. We order selectors according to the size of their conflict sets in decreasing order, and allocate the colors by considering the

[1] This does not matter on some processors that cannot deal easily with bytes or 16-bits words, such as the Dec Alpha (see [Sit92])

[2] This code was obtained from the compilation of C code, the order of the assembler operations being changed for clarity purposes.

```
! find in first argument array the offset associated with first target argument
ld     [%i0+4],%o0           ! o0=type number of 1st argument
sethi  %hi(_aa1),%o2
or     %o2,%lo(_aa1),%o2     ! o2=base address of arg-array 1
lduh   [%o0+%o2],%o0         ! o0=offset in dispatch table
! find in second argument array the offset associated with second target argument
ld     [%i1+4],%g1           ! g1=type number of 2nd argument
sethi  %hi(_aa2),%l1
or     %l1,%lo(_aa2),%l1     ! l1=base address of arg-array 2
ldub   [%g1+%l1],%g1         ! g1=ωγ₂π₂ =offset in dispatch table
! add offsets
add    %o0,%g1,%g1           ! g1=global offset in table (n − 1 additions)
! find code address
sethi  %hi(_meth),%o4
or     %o4,%lo(_meth),%o4    ! o4=table base address
ld     [%o4+%g1],%g1         ! g1=address of code to execute
jmpl   %g1,%g0               ! jump to the code
```

Figure 17: Sparc Implementation of the Dispatch Routine Using Dispatch Tables

selectors in this order. For a given selector, we first search in the list of colors ordered by creation time from the earliest, a color for which selectors do not conflict with the selector being considered. If no such color exists, a new color is created and associated with the selector. Following [AR92], we only consider, in the sets of applicable types, the types which do not have any subtypes.

As some argument-arrays use short (8-bits) words, and others long (16-bits) words, the coloring algorithm distinguishes the corresponding selectors. Thus, we do not group two argument arrays whose elements do not have the same size.

It often happens that no overriding occurs on some arguments of a generic function m: for some position i, the set of i-th formal arguments types is a singleton. In this case, the MSA method selection does not depend on the run-time type at this position, hence there is no corresponding dimension in the dispatch table. These selectors are called *inactive selectors*, the others being called *active selectors*. The associated formal argument is the single element of $Pole^i_m$, called a *single pole*.

As a consequence, the argument-arrays of inactive selectors may only be useful for type checking, to efficiently differentiate the subtypes of the single pole from the other types. The coloring of these arrays must only yield the selector arrays. Also,note that if two selectors have the same type as single pole, they can share the same selector array. We do not detail the construction of these selector arrays of inactive selectors, as they are not needed for run-time dispatch. Hence in the following, "selector" means "active selector", unless specified otherwise.

Consequently, our dispatch table computation algorithm with coloring is as follows :

1. Compute and store all poles. For each generic function m, for each argument position i of m, compute $Pole_m^i$ and store it, as well as the value of function $pole_m^i$, in a temporary array.

2. Create active selectors. This comes down to find the selectors with at least two poles.

3. Prepare argument-arrays fill-up. For each active selector (m, i), compute the corresponding $\omega \times \gamma_i^m$, and the corresponding breadth (8 bits or 16 bits). It is 16 bits if $\omega \times \gamma_i^m \times (|Pole_m^i| - 1) > 255$, because $(|Pole_m^i| - 1)$ is the highest pole number.

4. Build conflict graph for coloring. For each selector, find the applicable types which have no subtypes, by propagating the selector down the subtyping graph using a recursive function. The conflict set of each type without subtypes T is composed of the selectors applicable to T. For each such conflict set, add each selector to the conflict set of the other selectors.

5. Order the selectors by size of the conflict set, from the biggest one.

6. Compute the colors. A color is a set of selectors, and the color list is empty at the beginning. For each selector (m, i) of the list, look for a compatible color in the list of colors. A compatible color is composed of selectors which do not conflict with (m, i), and which have the same breadth as (m, i). If no such color exists, create a new color and add it to the list, then add (m, i) to this color. As an optimization, two color lists can be built, respectively for 8-bits selectors and 16-bits selectors.

7. Create the colored argument arrays, selector array, and associate each color with its colored argument array and selector array.

8. Identify selectors. For each color, sequentially number the selectors of the color.

9. Fill up the argument arrays. For each selector (m, i), using the temporary array of the function $pole^i_m$, for each type T such that $pole^i_m(T)$ is not null, π_i being the pole number of $pole^i_m(T)$, store $\omega \times \gamma^m_i \times \pi_i$ at T's place in the colored argument-array of (m, i), and store (m, i)'s number in the associated selector array.

10. Fill up the dispatch table, using the algorithm of Figure 14.

As coloring differentiates active selectors with 8-bits argument-arrays from those with 16-bits argument-arrays, the width of argument-arrays must be computed before coloring. This involves computing the number of poles in all dimensions. As coloring takes into account all selectors of all generic functions, the pole computation has to be done for all of these selectors. The argument-arrays are filled up before the dispatch table because, for each cell of the latter, we need to use the values of the preceding cells, based on Theorem 6. The argument-arrays are needed to quickly access these cells.

7.3 The Cecil Hierarchy

Our results have been conducted on the Cecil compiler [Cha93][3], which is a real object-oriented application with multi-methods, as this compiler is written in Cecil itself. This application includes 932 types, and 3990 generic functions, composed of 7873 methods.

Most of these functions have 0 or 1 active selector. Indeed, despite many generic functions have many arguments, it often occurs that their type is the same for all methods of the generic function. We say that an argument of a generic function is a target only if its selector is active. Note that in the Cecil terminology, the i-th argument of a generic function m is a *dispatching* argument iff $Pole^i_m$ includes at least one type which is not *any*, the common superclass of all classes. Hence, the set of target arguments is a subset of the set of dispatching arguments. In our view, dispatching arguments which are not target arguments are only considered for type-checking. No type checking is necessary for non-dispatching arguments, because all types are subtypes of *any*.

[3] The lists of classes and methods of the Cecil compiler were graciously put at our disposal by Craig Chambers and Jeff Dean from University of Washington

The table in Figure 18 gives the number of generic functions and methods respectively with 0, 1, 2, 3 and 4 target arguments. There are 305 types appearing as unique formal argument for a dispatching argument of a generic function.

Number of targets	Number of generic functions	Number of methods
0	2.761	2.761
1	1.147	4.465
2	75	586
3	6	59
4	1	2

Figure 18: Distribution of Generic Functions and Methods

7.4 Measurements

We first give the results that show the effectiveness of compression in terms of memory space, and computation time for the algorithms that construct dispatch tables.

7.4.1 Size of Dispatch Tables

The table in Figure 19 gives the total memory size in bytes of uncompressed, and compressed dispatch tables, for the generic functions with 2, 3 and 4 target arguments. We assume a system with memory addresses using 32-bits words. These totals take into account all generic functions, and as there are fewer generic functions with 3 target arguments than 2, the total size of the corresponding compressed tables is also smaller.

Number of Active Selectors	Size of Uncompressed Tables	Size of Compressed Tables
2	248,5 MB	7.648 B
3	18,1 GB	3.092 B
4	11 TB	64 B

Figure 19: Sizes of Dispatch Tables

7.4.2 Effectiveness of Coloring

We first consider generic functions with at least two target arguments, then generic functions with one target argument. Regarding the former, we distinguish the argument-arrays depending on their width, i.e. 8-bits or 16-bits. We also consider mono-targetted generic functions in order to give the total size of dispatch structures for the whole application.

	Uncolored Argument Arrays		Colored Argument-Arrays	
Multi-Targetted	on 8 bits	on 16 bits	on 8 bits	on 16 bits
Generic Functions	159	18	66	7
Mono-Targetted Gen. Fun.	1147		99	

Figure 20: Number of Argument Arrays

7.4.3 Total Size of Dispatch Structures

The total size of the structures needed for run-time dispatch tables includes the compressed tables (Figure 19), the colored argument arrays (Figure 20), the selector arrays (in case of dynamic type-checking) and the dynamic dispatch code (as shown in Figure 17). Assuming a 32-bits system, we summarize the results in Figure 21 in the case of multi-targetted generic functions, mono-targetted ones, and all generic functions. For mono-targetted generic functions, we assume mono-dimensional dispatch tables. The dispatch code is replicated either for each generic function if type checking is used, or for each colored dispatch table otherwise.

	Static type-checking	Dynamic type-checking
Multi-Targetted Generic Functions	89,788	161,264
Mono-Targetted Generic Functions	371,052	507,220
Total	460,840	668,484

Figure 21: Size of Dispatch Structures (in bytes)

7.4.4 Summary

We have described an optimized implementation of our multi-method dispatch scheme. Ignoring arguments that do not take part in the dispatch process saves both space and time. Storing the offsets of the cells in the argument-arrays, instead of pole numbers, ensures true constant-time dispatch. Replicating the dispatch code for each generic function saves some instructions at run-time. Using the minimal memory word width for argument-arrays, and grouping the argument-arrays that have the same color, optimizes the memory space needed by argument-arrays with no extra cost at run-time. This is of particular importance, as argument-arrays are significantly larger than any of the other structures. In the case of multi-targetted generic functions without type-checking, they represent 83% of the total 89,788 bytes. The structures used for type-checking are another kind of argument-arrays. Argument-arrays could be further optimized in the following ways:

- ordering optimally the argument positions of each generic function to increase the number of 8-bits wide argument-arrays.
- replacing coloring with a better compression scheme for bi-dimensional tables. An example is the "minimized row displacement" scheme of [DH95].
- grouping identical argument-arrays. If two selectors have the same formal arguments, the associated *pole* functions are then identical, hence their argument-arrays can be made identical, by defining the corresponding argument positions as the last of their respective generic functions. Indeed, the associated dimensions in the tables are then represented by low-level arrays, and the offsets (stored in the arguments-arrays) coincide with pole numbers.

On the opposite, the n-dimensional compressed tables only take 10804 bytes, a compression ratio of more than 99.99% w.r.t. Figure 19, considering only generic functions with 2 targets. Altogether, the compressed structures take a very small fraction of the memory space required by the uncompressed structures.

8 Related Work

8.1 Mono-Method Dispatch and Generalizations

Several techniques have been proposed to optimize the dynamic dispatch of mono-methods. Their presentation is relevant here because first they can be related to some of the techniques we used in our algorithms, and second they have been generalized to multi-method dispatch. We classify these techniques depending on whether they guarantee that method selection is done in constant or non-constant time. The latter techniques are mainly used in dynamically typed pure object-oriented languages. They try to optimize the average method dispatch time. Constant-time techniques are mostly used in statically typed non-pure object-oriented languages.

8.1.1 Non-Constant Time Techniques: Caching and Inlining

The technique known as *caching* consists in memorizing the MSA methods found by previous searches into a *cache*. The cache is then searched first and if a method is not found then the MSA is retrieved using the schema search approach. Several caching schemes exist and systems typically use a combination of them: a single global cache in Smalltalk and Sather [Deu83, UP83, SO91], local caches (local to a generic function or an invocation) in CLOS, PCL and Sather [KR90, SO91], inline caches in Smalltalk and Self [DS84, Ung86, UP87, CUL89, HCU91]. [KR90] proposes to extend its cache per generic function scheme to multi-method dispatch. Hashing is applied to the types of all the arguments, instead of a single argument, to access a cache entry.

The second technique, known as *inlining*, performs type analysis and run-time type tests to avoid method dispatch and *inline* the code of the MSA method. Several sophisticated techniques exist to achieve inlining: type prediction [UP83, GR83, DS84] and type casing [REJZ88] in Smalltalk, customization, splitting [CUL89, CU91, USCH92], type feedback and adaptive optimization [HU96] in Self.

8.1.2 Constant Time Techniques: Dispatch Tables

In the case of mono-targetted generic functions, dispatch tables are bi-dimensional, and can be organized in three ways. The first one is a unique global two-dimensional array with the types and the generic functions as indices. The second organization associates to every generic function a one-dimensional array indexed by the types with an entry for every type. The third organization associates to every type a one-dimensional array indexed by the generic functions. The last two organizations are just two different ways of slicing the global table. Independently of the organization, the number of entries in the dispatch table is $|\Theta| \times |F|$, hence in practice the tables need to be compressed. For example in a Smalltalk-80 system, $|\Theta| \times |F|$ amounts to about 3.5 millions entries.

In the case of single-inheritance C++, each type owns a single method table (the *_vtbl*). Finding the MSA method requires to get the base address of the dispatch table stored in the target argument, and perform one array access using the index of the generic function. This amounts to two indirections and one addition. This scheme eliminates all empty cells, and was adapted when multiple inheritance was added to C++[4]. If a class C inherits classes A and B, C's instances are composed of two parts, which correspond to the properties respectively inherited from A and B. Because of polymorphism, each of these parts begins with references to distinct dispatch tables, which are used when the instances of C are considered as A's or as B's instances. This scheme does not seem to generalize well to multi-methods. Indeed, the method tables would then be associated with n-tuples of classes, and in case of multiple inheritance each n-tuple of objects should be split in several parts, which is impossible since each objet should have several cut-outs.

Coloring was originally used to compress the bi-dimensional dispatch tables of mono-methods. Lines are labeled with generic functions, and columns with types. As for the merging of argument-arrays described in Figure 4, coloring consists of merging lines in which non-empty cells are not associated with the same type. As an extension, [HC92] proposes to merge both lines and columns of the dispatch tables. *Row displacement* [Dri93, DH95] also proceeds by merging lines and columns. Before being merged, these lines and columns are

[4] The cost of dynamic dispatch may then rise to three indirections and two additions.

shifted by a variable number of cells. [DH95] shows that row displacement is more efficient on lines than on columns. Experimentally, the best compression factor obtained is a factor of 66. These techniques might be used to compress multi-method dispatch tables, but they only aim at eliminating empty cells. Note however that row displacement could be used to compress our argument-arrays, possibly yielding a better compression rate.

Besides, [VH94] also proposes to group similar columns of mono-method dispatch tables. When two corresponding cells differ between two similar columns, the resulting cell in the merged column contains the address of an intermediary routine that chooses the adequate method at run-time. This technique however does not offer constant-time selection. Moreover, [VH94] only considers single inheritance.

Finally, [Que95] proposes to use a different structure than tables to store the pre-calculated MSA method of each class. It consists of decision trees, composed of three kinds of nodes, traversed by dynamic dispatch using the class of the target at run-time. "`cst`" nodes hold one method reference; they can be nested in "`if`" nodes to group the classes having the same MSA method as a common superclass. Finally, "`xif`" nodes hold arrays that relate each subclass of a class to its MSA method. These trees hence have a limited number of empty cells, and enable to group some cells with the same contents. Each node of the decision tree is used in constant time at run-time, but if the decision tree is not balanced, dynamic selection is not performed in constant time. [Que95] recognizes that many decision trees can be associated with a given generic function. However, it seems that finding the smallest tree requires to build all possible trees and compare them. Finally, this proposal does not take into account multiple inheritance and multiple targetting.

8.2 Dispatching Using Lookup Automata

In [CT95], the authors present algorithms to dispatch multi-methods using some sort of "dispatch trees"[5], which as our dispatch tables, hold the precalculated MSA method for each possible invocation of a generic function. Their multiple-dispatching scheme offers a quasi-constant time. Furthermore, dispatch trees are compressed using a notion very similar to our poles.

[5] Dispatch trees is our own terminology to describe what the authors call a lookup automaton

The formalism presented in [CT95] is very different from ours, and its exposure is quite long. Thus, we first present our own description of the principles of their algorithms and then draw a comparison between their algorithms and ours.

8.2.1 General Principle

The *dispatch tree* of a n-ary generic function m is a directed, balanced tree of depth n. Each invocation signature is associated with a unique path in this tree. Each path starts from the root, and each type T_i of the signature determines the choice of one of the possible branches of the tree to expand the path of length $i-1$ into a path of length i. The leaves of the tree are labeled with the MSA method of the invocation.

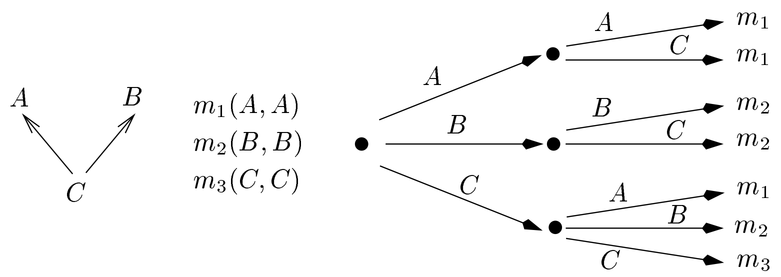


Figure 22: Type Graph and Generic Function With Dispatch Tree

Example 8.1: Consider in Figure 22 the graph of types A, B and C, the generic function m, and the dispatch tree associated with m. The arcs are labeled with types. Each signature for which there is an MSA method is associated with a path in the tree. This path leads to a leaf labeled with the corresponding MSA method. An invocation with signature (A, C) corresponds to a path starting from the root that follows the arcs labeled by A and C, yielding an MSA method m_1. An invocation with signature (A, B) has no associated path since $m(A, B)$ has no MSA method. □

There is a single path $T_1, T_2, \ldots, T_{i-1}$ that starts from the root and reaches a node N_i of depth i. Furthermore, from node N_i, there is exactly one output arc for each type $T_i \in \Theta$ such that there exists an invocation $m(T_1, \ldots, T_i, \ldots)$ for which there is an MSA method.

In fact, since an invocation $m(T_1,\ldots,T_n)$ has the same MSA method as $(T'_1,\ldots,T'_n)$, where $\forall i,\ T'_i \in Pole^i_m$, arcs of the tree can be associated with poles only. Moreover, suppose that a node N of depth i is reached by a path $T'_1,\ldots,T'_{i-1}$ where each T'_i is an i-pole, the set of methods that are applicable to an invocation where the $i-1$ first types are $(T'_1,\ldots,T'_{i-1})$ forms a subset $\mathcal{M}'$ of the set of methods associated with m. The subtree with root N only needs to explore this set $\mathcal{M}'$ of applicable methods. Thus, the number of output arcs for N is determined by the number of i-poles that occur at the ith position in the methods of $\mathcal{M}'$. We call these i-poles the *local poles* of N, and their set is noted $Pole^N$. [Duj96] gives a formal definition of these notions.

Example 8.2: In the tree of Figure 22, all types on the labels of arcs are poles. Consider the two nodes N_1 and N_2 reached by signatures respectively starting with A and B at the first position. The set of possible applicable methods to these signatures is $\{m_1, m_2\}$. The set of 2-poles is $\{A, B, C\}$. However, $Pole^{N_1} = \{A\}$ and $Pole^{N_2} = \{B\}$. Thus, the output arcs of N_1 and N_2 labeled by C can be discarded from the tree. □

A further compression can be obtained by *node unification*. Two nodes N_1 and N_2 of a given depth can be unified into one node N, if it is possible to superpose the two subtrees with respective roots N_1 and N_2. This superposition must take into account the types labeling the arcs and the MSA methods that label the leave nodes. Then the arcs that lead to N_1 and N_2 lead to N, and the dispatch tree becomes a direct, acyclic graph. This unification spares the space taken by one node and all its subtree.

Example 8.3: Considering the dispatch tree of Figure 23, two unifications can be done. The first one unifies the nodes N_1 and N_2, and the second unifies N'_1, N'_2 and N'_3. This spares 6 nodes. □

8.2.2 Comparison

The approach of [CT95] supports languages which precedence ordering is Inheritance Order Precedence, as defined in [ADL91]. This order is a particular case of Argument Subtype

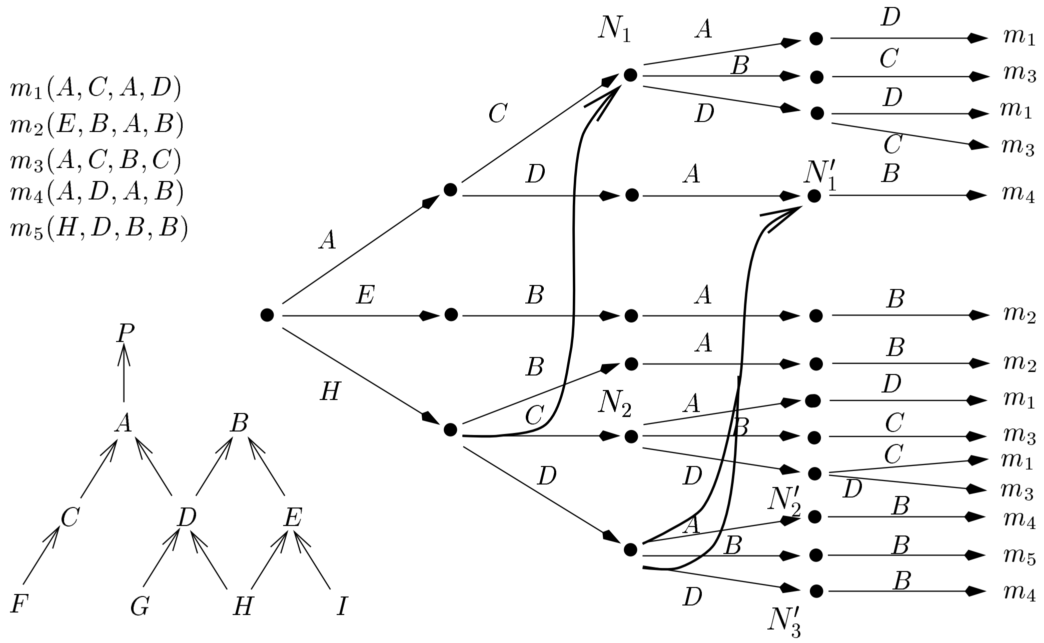


Figure 23: Dispatch Tree with 4 Arguments

Precedence with monotonicity, considered throughout this report. No implemented language currently supports the former order[6], while both Cecil and Dylan support the latter.

The proposal of [CT95] includes a pole computation algorithm (called there a closure algorithm). This algorithm does not compute the influences. It operates in two steps. The first step duplicates the type graph. The complexity of this step is not reported. The second step traverses the type graph from the most generic to the most specific types. In this traversal, each type which is found not to be a pole is removed from the graph, and its direct subtypes become direct subtypes of its pole. This way, the poles of the supertypes appear as direct supertypes. In contrast, our proposal associates a pole with each type, which avoids to update the type graph.

[6] Inheritance Order Precedence was presented in [ADL91] to model the precedence algorithm of CLOS, however the former is monotonic by definition, while the latter is not, as shown in [DHHM92]

For each type, their traversal computes the most specific supertype poles. The technique used for subtyping comparisons is that of [ABJ89]. This technique associates with each type a sequential number, and the list of its supertypes's numbers. Testing if T_1 is a subtype of T_2 then comes down to look for T_2's number in T_1's list. The experimental complexity of this technique is described in [ABJ89] as "essentially constant". [CT95] base the complexity of their pole computation algorithm on this experimental complexity. However, the worst-case complexity of this technique is $O(log|\Theta|)$. Indeed, each test implies a loop through a list, which can be long if T_1 has many supertypes. Our proposal is different due to the representation of the collection of supertypes, which allows to perform the test with a direct access in a small and constant number of basic operations.

To summarize, the complexity of their algorithm is $\mathcal{O}(|\Theta| + \mathcal{E})$. However, this does not take into account the copy of the type graph, and the worst-case complexity of the subtype comparison.

The construction of the dispatch tree requires more pole computations than the dispatch table. Indeed, one pole computation must be done for each node. The number of nodes is bounded by $\sum_{i=1}^{n} |\Theta|^i$. In contrast, the number of pole computations in a dispatch table is n. Moreover, node unification also takes supplementary time in dispatch trees. On the other hand, filling up the dispatch tables requires more MSA computations than with dispatch trees. Finally, as we do not assume the same precedence ordering to compute the MSA methods, it is difficult to compare the effective construction time.

The nodes of the dispatch trees are implemented as two arrays, which respectively hold the local poles and the succeeding nodes of each local pole. Run-time dispatch traverses the dispatch tree using the types of the successive arguments of the invocation. Each type T is used against one node N to obtain the succeeding node in two steps. The first step looks for the pole of T in the array of local poles of N. If this pole is found at position k, the second step retrieves the succeeding node at position k in the array of succeeding nodes. The first step is done in time linear to $|Pole^v|$, by finding the first pole supertype of T. To keep this time bounded by a constant, nodes have a different structure when the number of local poles is greater than an arbitrary constant c. This structure is then a single array that associates with each type of the schema its successor node, which is found in constant-time.

The time needed for dynamic dispatch hence obviously depends on c. If $|Pole^v| \leq c$, the time complexity of finding the successor vertex is $O(|Pole^v|)$ (assuming that the subtyping tests are done in constant time). If $|Pole^v| > c$, this time is a constant. $|Pole^v|$ is bounded by $|\Theta|$, hence the worst-case time complexity of dynamic dispatch is $O(n \times |\Theta|)$. In contrast, the time for dispatch tables is $n \times \gamma$, γ being a constant.

The memory space taken by dispatch trees also depends on c. If c is high, dispatch trees benefit from the progressive decrease of the number of local poles, and the absence of argument-arrays. However, dynamic dispatch is guaranteed to be done in constant time only if c is 1. Then many nodes have a size of Θ words. This is equivalent to having many argument-arrays for each generic function, and in this case the dispatch trees take a lot more space than dispatch tables.

In [Duj96], we present dispatch trees that offer constant-time dispatch. In that study, trees offer better compression rates than tables only for $n \geq 3$. Tri-dimensional compressed tables however do not seem to take a significant space. Thus, dispatch trees should be considered only for $n \geq 4$, and with enough methods. We also observed that tree construction was significantly slower than table construction, due to the number of pole computations.

9 Conclusion

We proposed a simple multi-method dispatch scheme based on compressed dispatch tables. The salient features of our proposal are the following. First, it guarantees a dynamic dispatching in constant time, an important property for some object-oriented applications. Second, unlike the proposals such as [CT95], it is applicable to most existing object-oriented languages, both statically and dynamically typed, since it only assumes that method selection does not contradict argument subtype precedence, a most common property of method orderings. Last, our scheme is simple to implement and quite effective: in most cases, it yields a minimal dispatch table [AGS94].

We provided efficient algorithms to obtain the structure of a dispatch table in a linear time in the number of types, and to fill it up with MSA methods's addresses. Our measurements show that the compression of dispatch tables is very effective, resulting in a very small fraction of the space required by the uncompressed structures. Further compression can be obtained at the expense of implementation complexity and increased compile time, using dispatch trees [Duj96] (for 4 targets or more) or better schemes of argument-arrays compression, such as row displacement [DH95] or identical array grouping.

Several issues are left for future work. First, it would be useful to quantify the effectiveness of our compression scheme on other real applications that use multi-methods. Second, in [AGS94], we presented another possible optimization, which consists of sharing compressed dispatch tables between generic functions. A study of multi-methods's definition patterns in real applications would allow the development of heuristics to guide the use of this optimization. Third, applying our scheme to interactive programming environments requires to develop incremental versions of our algorithms. These incremental algorithms should also take care of method disambiguation, as presented in [AD96].

Acknowledgments: Special thanks go to Olivier Gruber for his contribution to the earlier work on dispatch tables. We would also like to thank Pascal André, Marie-Jo Bellosta, Michel Habib and Patrick Valduriez for their insightful comments on earlier versions of this paper.

References

[ABDS96] Eric Amiel, Marie-Jo Bellosta, Eric Dujardin, and Eric Simon. Type-safe relaxing of schema consistency rules for flexible modelling in OODBMS. *VLDB Journal*, 5(2), April 1996.

[ABJ89] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *ACM SIGMOD Conference Proceedings*, 1989.

[AD96] E. Amiel and E. Dujardin. Supporting explicit disambiguation of multi-methods. In *ECOOP Conference Proceedings*, 1996.

[ADL91] R. Agrawal, L. G. DeMichiel, and B. G. Lindsay. Static type checking of multi-methods. In *OOPSLA Conference Proceedings*, 1991.

[AGS94] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-methods dispatch using compressed dispatch tables. In *OOPSLA Conference Proceedings*, 1994.

[App95] Apple Computer. *Dylan Reference Manual, draft*, September 1995. Available from the Dylan Home Page on the World-Wide Web at http://www.cambridge.apple.com/dylan/dylan.html.

[AR92] P. André and J.-C. Royer. Optimizing method search with lookup caches and incremental coloring. In *OOPSLA Conference Proceedings*, pages 110–126, 1992.

[BDG+88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System specification. *ACM SIGPLAN Notices*, 23, Sept. 1988.

[BKK+86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In *OOPSLA Conference Proceedings*, 1986.

[Cha92] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP Conference Proceedings*, 1992.

[Cha93] C. Chambers. The Cecil language, specification and rationale. Technical Report 93-03-05, Dept of Computer Science and Engineering, FR-35, University of Washington, March 1993.

[CL95] C. Chambers and G. T. Leavens. Typechecking and modules for multimethods. *ACM TOPLAS*, 17(6), November 1995.

[CT95] W. Chen and V. Turau. Multiple dispatching based on automata. *TAPOS*, 1(1), 1995.

[CU91] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *OOPSLA Conference Proceedings*, 1991.

[CUL89] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF, a dynamically typed object-oriented language based on prototypes. In *OOPSLA Conference Proceedings*, pages 49–70, 1989.

[Deu83] L. P. Deutsch. *Smalltalk-80: Bits of History and Words of Advice*, chapter The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture. Addison Wesley, 1983.

[DH95] K. Driesen and U. Hölzle. Minimizing row displacement dispatch tables. In *OOPSLA Conference Proceedings*, 1995.

[DHHM92] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *OOPSLA Conference Proceedings*, 1992.

[DHHM94] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *OOPSLA Conference Proceedings*, 1994.

[DMSV89] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *OOPSLA Conference Proceedings*, pages 211–214, 1989.

[Dri93] K. Driesen. Selector table indexing and sparse arrays. In *OOPSLA Conference Proceedings*, 1993.

[DS84] L. P. Deutsch and A. Schifman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the 11th Annual ACM POPL Conference Proceedings*, 1984.

[Duj96] E. Dujardin. Efficient dispatch of multimethods in constant time with dispatch trees. Technical Report 2892, INRIA, 1996.

[ES92] M. A. Ellis and B. Stroustrup. *The annotated C++ reference manual.* Addison-Wesley, Reading, Mass., 1992.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison Wesley, Reading, MA, 1983.

[HC92] S.-K. Huang and D.-J. Chen. Two-way coloring approaches for method dispatching in object-oriented programming system. In *Proceedings of the Annual International Computer Software and Applications Conference*, 1992.

[HCU91] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically typed object-oriented languages using polymorphic inline caches. In *ECOOP Conference Proceedings*, pages 21–36, 1991.

[HU96] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM TOPLAS*, 18(4), July 1996.

[Knu73] D. Knuth. *The Art of Computer Programming, Fundamental Algorithms, Second Edition.* Addison-Wesley, 1973.

[KR90] G. Kiczales and L. Rodriguez. Efficient method dispatch in PCL. In *Proceedings of ACM Conference on Lisp and Functional Programming*, 1990.

[Mel94] J. Melton, editor. *(ISO Working Draft) SQL Persistent Stored Modules (SQL/PSM).* ANSI X3H2-94-331, August 1994.

[Mel96] J. Melton. An SQL3 snapshot. In *Intl. Conf. on Data Engineering Conference Proceedings*, 1996.

[MHH91] W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-methods in a statically-typed programming language. In *ECOOP Conference Proceedings*, 1991.

[Que95] C. Queinnec. Fast and compact dispatching for dynamic object-oriented languages. Submitted for publication, available by ftp on ftp.inria.fr as /INRIA/Projects/icsla/Papers/dispatch.ps, 1995.

[REJZ88] J. O. Graver R. E. Johnson and L. W. Zurawski. TS: An optimizing compiler for Smalltalk. In *OOPSLA Conference Proceedings*, pages 18–26, 1988.

[Ros88] J. R. Rose. Fast dispatch mechanism for stock hardware. In *OOPSLA Conference Proceedings*, pages 27–35, 1988.

[Sit92] R. L. Sites. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[SO91] H. W. Schmidt and S. Omohundro. Clos, Eiffel, and Sather: A comparison. Technical Report TR-91-047, International Computer Science Institute, Berkeley, CA, 1991.

[Ung86] D. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, Cambridge, MA, 1986.

[UP83] D. Ungar and D. Patterson. *Smalltalk-80: Bits of History and Words of Advice*, chapter Berkeley Smalltalk: Who Knows Where the Time Goes? Addison Wesley, 1983.

[UP87] D. Ungar and D. Patterson. What price Smalltalk? *IEEE Computer*, 20(1), 1987.

[USCH92] D. Ungar, R. B. Smith, C. Chambers, and U. Hölzle. Object, message and performance: How they coexist in SELF. *IEEE Computer*, October 1992.

[VH94] J. Vitek and R. N. Horspool. Taming message passing: Efficient method look-up for dynamically typed languages. In *ECOOP Conference Proceedings*, 1994.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399